Simple Input/Output Streaming in the *Roadrunner* Operating System*

Frank Miller, George Apostolopoulos, and Satish Tripathi

Mobile Computing and Multimedia Laboratory
Department of Computer Science
University of Maryland
College Park, MD 20742
{*fwmiller, georgeap, tripathi*}*@cs.umd.edu*

November 13, 1996

1 Introduction

On-demand multimedia applications can impose Quality-of-Service (QoS) parameters on streaming data transfers performed by operating systems. For multimedia servers, high performance in the form of low latency and high throughput are also necessary. This work seeks to improve the performance of *simple* data streaming between hardware devices with QoS in the context of general-purpose operating systems. Simple streaming refers to moving data from one device to another without transforming (e. g. compressing) the data during the transfer.

A transfer model originates data at a source device, traverses a transfer path, and delivers data to a destination device. In this work, we our interested in sources that are files on a mass storage device being served to a destination network adapter. For these transfers, the path must necessarily include the file system and network protocol stack.

Streams are partitioned into *burst* and *periodic* streams. Burst streams transfer data from source to destination as quickly as possible, one packet after another. FTP and HTTP file transfers fall into this category. There are typically no QoS parameters associated with this type of stream but they will be present in a general-purpose environment. Periodic streams transfer data based on some frequency, e. g. a frame rate for video. These streams impose Quality-of-Service (QoS) parameters like bounded *delay* and *delay variance* (i. e. jitter).

In traditional operating systems such as 4.4BSD [1], input/output (I/O) subsystems implement an environment that supports streaming through applications. System calls are provided to allow user applications to read data from and write data to devices. When an application program executes a `read()` system call on a file, data is transferred from the disk to a buffer in the kernel and then to the application. A subsequent `write()` system call to a network socket results in data being copied to an *mbuf* in the kernel and then to an network device. For applications that stream data using read() and write() in succession, such as FTP and web servers and more recently, on-demand audio and video servers, the cross domain data copies and number of system calls reduce potential performance.

A new operating system kernel and input/output subsystem have been designed and implemented to address simple streaming applications. The specific aim of the *Roadrunner* operating system is to support high-performance, concurrent, simple data streaming in the context of a general-purpose operating system environment. This goal is addressed with the `stream()` system call which allows flexible control over kernel data streaming.

Section 2 discusses the design and implementation of the *Roadrunner* operating system and

the `stream()` system call. Section 3 presents comparative measurements for a simple streaming application executing on *Roadrunner* (in its current form) and BSD. Section 4 presents conclusions and future work.

2 *Roadrunner* Design and Implementation

Roadrunner is designed around the need to add support for simple kernel streams to the general purpose operating system environment. In order to support concurrent, deterministic streaming, the following design elements have been incorporated.

Multi-threaded The kernel is multi-threaded to support *scheduled* concurrent activity within the kernel. Currently, each stream has a thread allocated to perform data transfers, however, a necessary optimization will unify stream processing under a single thread.

Real-time To support threads that delay and jitter constraints, a fully preemptable, statically prioritized kernel design is implemented. General-purpose computing loads are supported on a time available basis using lower priority values.

Unified I/O Programming Interface While the `stream()` system call could be designed to explicitly handle endpoints with different programming interfaces, such as sockets [5], a more elegant design unifies the interface to all the endpoints using the file system name space.

Unified Buffer Pool In order to avoid data copies between different buffer pool types, a unified buffer pool to be used by block device drivers, file systems, *and* the protocol stack will be utilized.

2.1 I/O Subsystem Architecture

Access to all I/O elements, files, device drivers, and network protocols is through the global file system interface using file system path name conventions. Underlying this interface are specific file system and network protocol implementations and direct access to the device drivers. Figure 1 illustrates the key architectural differences between BSD and *Roadrunner* .

When a path name is presented to the *Roadrunner* file system, the mount table is consulted to determine on which file system the path corresponds. The prefix is then stripped off and the remaining path is passed to the specific file system implementation. Table 1 describes how the `udpfs` file system implementation utilizes path name conventions to provide access to protocol communications endpoints.

Specific file systems and network protocols utilize a common buffer pool. The aim is to allow the `stream()` system call to pass references to buffers between specific file system implementations, reducing the number of data copies. This portion of the work requires significant effort to rework the use of buffers in the specific file system implementations, including the network protocols. This effort is under way.

2.2 Implementation Status

Roadrunner currently runs on IBM PC-compatibles. The kernel, several device drivers, the generic file system interface, and a DOS-compatible specific file system implementation have been written from scratch and are up and running. The DEC Tulip PCI Ethernet adapter device driver has been ported from FreeBSD. The UDP/IP/ICMP/ARP protocol stack has been ported from XINU [2]. A simple text based, virtual console user interface has been developed and is in place to allow system monitoring and debugging.

The network device driver and protocol stacks have been recently ported and still have some minor bugs that have prevented measurements under load at the time of this writing. However, *Roadrunner* is capable of streaming data from the hard disk to the ethernet. The next section describes our initial observations of the system's performance.

3 Performance

The focus of performance measurements is to determine the supported concurrent streaming load

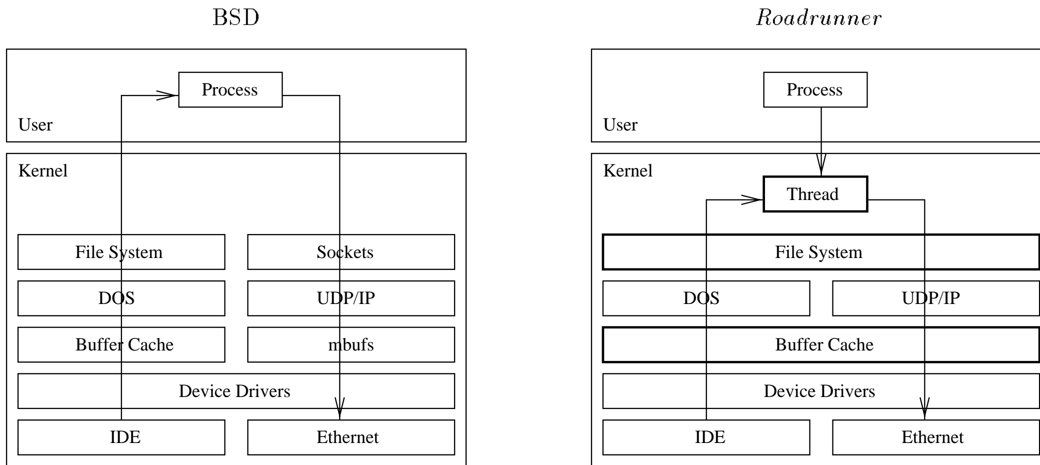

Figure 1: The I/O Architectures for BSD and *Roadrunner*

Table 1: UDP File Name Conventions

File Mode	File Name (`/net/udp`)	Description
`O_RDONLY`	`/4000`	Open local port 4000 for reading
`O_WRONLY`	`/2592`	Open port 2592 on local host (ip address = 127.0.0.1) for writing
	`/128.8.130.115/2592`	Open port 2592 on foreign host with ip address = 128.8.130.115 for writing
`O_RDWR`	`/2592`	Open port 2592 on local host (ip address = 127.0.0.1) for writing and pick an arbitrary local port number for reading
	`/4000/2592`	Open port 2592 on on local host (ip address = 127.0.0.1) for writing and local port 4000 for reading
	`/128.8.130.115/2592`	Open port 2592 on foreign host with ip address = 128.8.130.115 for writing and pick an arbitrary local port number for reading
	`/4000/128.8.130.115/2592`	Open port 2592 on foreign host with ip address = 128.8.130.115 for writing and local port 4000 for reading

for a given hardware platform. Measurements would span burst only loads to periodic only loads to a mixture of burst and periodic loads.

While a large set of measurements has been taken for BSD, the stability of our current implementation has not allowed us to complete the same set of measurements for *Roadrunner* at this time. However, *Roadrunner* is capable of streaming data from the hard disk to the ethernet. Figure 2 presents an initial set of comparison measurements for BSD and *Roadrunner* .

BSDi, version 2.1 and *Roadrunner* were used as servers with a BSDi, version 2.1 platform serving as client for both streaming tests. Both servers were run on an IBM PC/350 with 75Mhz Pentium, 16 Mbytes main memory, 256 Kbyte second level cache, 810 Mbyte EIDE hard drive and SMC Etherpower 10 BaseT PCI Ethernet Adapter. A 110 Mbyte MPEG-1 encoded video [1] resident on a DOS file system partition was used as the source file for all streams.

Each server ran a single periodic stream that read a 1 Kbyte packet from the file just described and wrote the packet to a UDP port at a rate of 30 packets per second (i. e. a period of approximately 33,333 microseconds). Such a stream models the transmission of a small (perhaps 320x200) MPEG video. Figure 2 gives the latency of each of the first 10,000 (of approximately 106,000) packets. Latency was measured from the beginning of each period to the end of the UDP write.

There are two points to observe. First, the BSD figures display a set of diagonal patterns of points due to difficulty implementing the real-time loop. The `usleep()` call used to implement the delay required to wait for the beginning of the next period is highly inaccurate. This is not suprising, BSD is not intended to be a real-time system.

Second, the *Roadrunner* DOS file system implementation does not currently use a buffer cache. Each open file descriptor stores a single cluster from the disk and when it is exhausted, reads the next cluster. Despite this implementation deficiency, the system displays remarkably good raw performance.

[1] This video is an MPEG-1 encoding of approximately the first 20 minutes of the James Bond move, "Live and Let Die".

4 Related Work

The design of the `stream()` system call was heavily influenced by Kevin Fall's work at UCSD [3, 4]. The `splice()` system call was proposed for addition to the UNIX I/O subsystem. The idea was to supplement what was termed a *memory-oriented I/O* (MIO) model with a kernel based *peer-to-peer I/O* (PPIO) model. These terms are equivalent to the push-pull and stream models, respectively, described in this work. The UCSD work demonstrated that significant performance improvements can be achieved when cross-address space data copies and system calls are reduced. Our work seeks to improve on this result by supporting QoS parameters.

5 Conclusion

While a significant amount of work has been done already, much remains. Most important is the design and development of the unified buffer cache. There are a number of issues to be addressed.

- What policy should be used for flushing buffers from the cache?

  Some global information about which buffers (being used by different specific file systems) can be freed and reused will be necessary.

- How are differing buffer size needs handled?

  Different specific file system implementations will have different optimal buffer sizes. For example, the BSD FFS implementation transfers data from disk in blocks of 2048 bytes, ethernet packets used in Internet protocol stacks are limited to 1500 bytes, ATM cells are only 53 bytes. Some investigation into how to manage buffers with respect to these different buffer size needs is necessary.

- How is prepending of data to a buffer handled without data copies?

  BSD *mbufs* handle this problem quite well. A similar approach may be necessary for *Roadrunner* buffers as well.

There is a need for more determinism in the end system operating systems performing streaming data transfers due to impending QoS requirements. With this in mind, this work

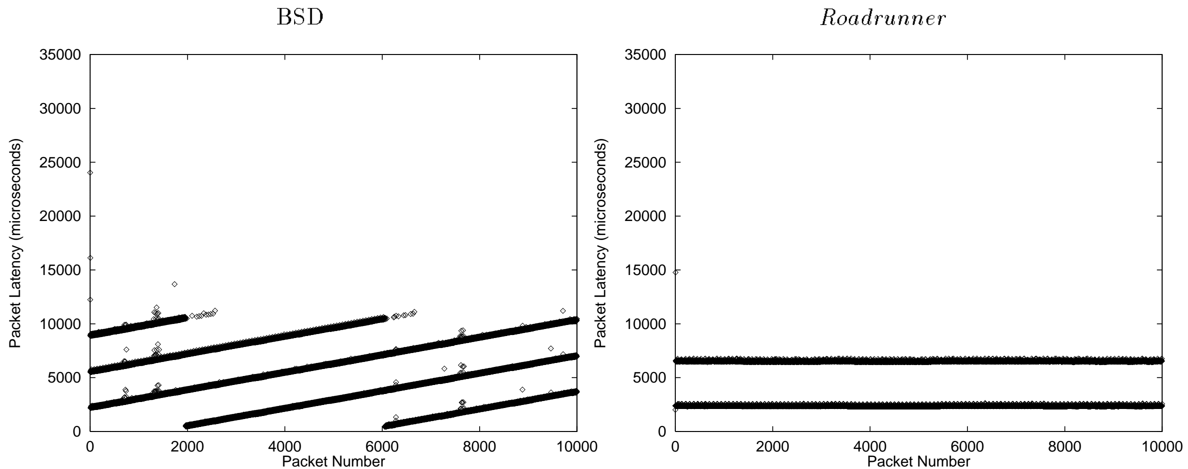

Figure 2: Sample Stream Transmission Traces

attempts to provide improved QoS under load for streaming applications running in a general-purpose operating system environment.

Comparisons to BSD are relevant since it is used heavily in the current Internet environment. It appears from the scant initial measurements presented here and those we have taken that were not presented, that more attention to QoS parameters will be required if multimedia applications like on-demand audio and video are to be supported efficiently.

References

[1] McKusick, M., Bostic, K., Karels, M., and Quarterman, J., *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, 1996.

[2] Comer, D. and Stevens, D., *Internetworking with TCP/IP: Volume II*, Prentice-Hall, 1994.

[3] Fall, K. and Pasquale, J., "Improving Continuous-Media Playback Performance With In-Kernel Data Paths", Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS), pp. 100-109, 1994.

[4] Fall, K., *A Peer-to-Peer I/O System in Support of I/O Intensive Workloads*, Ph. D. Thesis, University of California/San Diego, 1994.

[5] Stevens, W. R., *TCP/IP Illustrated, Volume 2 - The Implementation*, Addison-Wesley, 1994.