STREAMS-based vs. Legacy Pipe Performance Comparison

Experiment Test Results for Linux

Brian F. G. Bidulock*
OpenSS7 Corporation

July 26, 2008†

Abstract

With the objective of contrasting performance between STREAMS and legacy approaches to system facilities, a comparison is made between the tested performance of the Linux legacy pipe implementation and the STREAMS-based pipe implementation using the *Linux Fast-STREAMS* package [LfS].

1 Background

Pipes have a rich history in the UNIX operating system. Present on early Bell Laboratories UNIX Versions, pipes found their way into both BSD and System V releases. Finally, in 4.4BSD pipes are implemented with Sockets and in System V Release 4 pipes are implemented with STREAMS.

1.1 STREAMS

STREAMS is a facility first presented in a paper by Dennis M. Ritchie in 1984 [Rit84], originally implemented on 4.1BSD and later part of *Bell Laboratories Eighth and Ninth Edition UNIX*, incorporated into *UNIX System V Release 3* and enhanced in *UNIX System V Release 4* and further in *UNIX System V Release 4.2*. STREAMS was used in SVR4 for terminal input-output, pseudo-terminals, pipes, named pipes (FIFOs), interprocess communication and networking. Since its release in *System V Release 3*, STREAMS has been implemented across a wide range of UNIX, UNIX-like and UNIX-based systems, making its implementation and use an ipso facto standard.

STREAMS is a facility that allows for a reconfigurable full duplex communications path, *Stream*, between a user process and a driver in the kernel. Kernel protocol modules can be pushed onto and popped from the *Stream* between the user process and driver. The *Stream* can be reconfigured in this way by a user process. The user process, neighbouring protocol modules and the driver communicate with each other using a message passing scheme. This permits a loose coupling between protocol modules, drivers and user processes, allowing a third-party and loadable kernel module approach to be taken toward the provisioning of protocol modules on platforms supporting STREAMS.

On *UNIX System V Release 4.2*, STREAMS was used for terminal input-output, pipes, FIFOs (named pipes), and network communications. Modern UNIX, UNIX-like and UNIX-based systems providing STREAMS normally support some degree of network communications using STREAMS; however, many do not support STREAMS-based pipe and FIFOs[1] or terminal input-output[2] directly or without reconfiguration.

1.2 Pipe Implementation

Traditionally there have been two approaches to implementation of pipes and named pipes (FIFOs):

Legacy Approach to Pipes.

Under the 4.1BSD or SVR3 approach, pipes were implemented using anonymous FIFOs. That is, when a pipe was opened, a new instance of a FIFO was obtained, but which was not attached to a node in the file system and which had two file descriptors: one open for writing and the other opened for reading. As FIFOs are a fundamentally unidirectional concept, legacy pipes can only pass data in one direction. Also, legacy pipes do not support the concept of record boundaries and only support a byte stream. Each end of the pipe uses a the legacy interface and they do not provide any of the advanced capabilities provided by STREAMS.

SVR4 Approach to Pipes.

Under the SVR4 approach, both pipes and FIFOs are implemented using STREAMS [GC94]. When a pipe is opened, a new instance of a STREAMS-based pipe is obtained, but which is attached to a non-accessible node in the `fifofs` file system instead of the normal STREAMS `specfs` file system. Although one file descriptor was opened for read and the other for write, with a STREAMS-based pipe it is possible to reopen both for reading and writing.

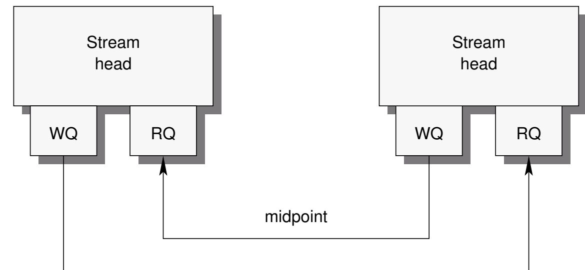

Figure 1: STREAMS-Based Pipes

The STREAMS-based pipes provide the same rich set of facilities that are also available for other STREAMS devices such as pseudo-terminals and network interfaces. As a result, STREAMS-based pipes provide a number of capabilities that are not provided by legacy pipes:

Full Duplex. STREAMS-based pipes are full duplex pipes. That is, each end of the pipe can be used for reading and writing. To accomplish the same effect with legacy pipes requires that two legacy pipes be opened.

Pushable Modules. STREAMS-based pipes can have STREAMS modules pushed an popped from either end of the pipe, just as any other STREAMS device.

File Attachment. STREAMS-based pipes can have either end (or both ends) attached to a node in the file system using fattach(3) [Ste97].

File Descriptor Passing. STREAMS-based pipes can pass file descriptors across the pipe using the `I_SENDFD` and `I_RECVFD` input-output controls [Ste97].

*bidulock@openss7.org

†Original edition June 16, 2007

1. For example, AIX.
2. For example, HP-UX.

Record Boundary Preservation. STREAMS-based pipes can preserve record boundaries and can pass messages atomically using the getmsg(2) and putmsg(2) system calls.

Prioritization of Messages. STREAMS-based pipes can pass messages in priority bands using the getpmsg(2) and putpmsg(2) system calls.

BSD Approach to Pipes.

As of 4.2BSD, with the introduction of Sockets, pipes were implemented using the networking subsystem (UNIX domain sockets) for what was cited as *"performance reasons"* [MBKQ97]. The pipe(2) library call effectively calls sockpair(3) and obtains a pair of connected sockets in the UNIX domain as illustrated in *Figure 2*.

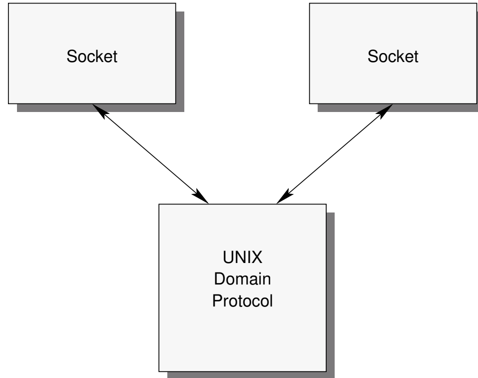

Figure 2: 4.2BSD Pipes

Knowing the result of this testing, I can only imagine that the *"performance reasons"* had to do with the lack of a full flow control mechanism in the legacy file system based pipes.

Linux Approach to Pipes.

Linux adopts the legacy (4.1BSD or SVR3 pre-STREAMS) approach to pipes. Pipes are file system based, and obtain an `inode` from the `pipefs` file system as illustrated in *Figure 3*. Pipes are unnamed FIFOs, unidirectional byte streams, and do not provide any of the capabilities of STREAMS-based pipes or socket pairs in the UNIX domain.[3]

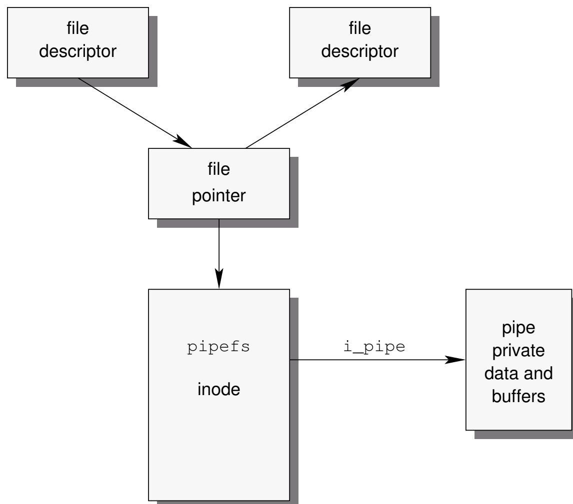

Figure 3: Linux Legacy Pipes

Standardization.

During the POSIX standardization process, pipes and FIFOs were given special treatment to ensure that both the legacy approach to pipes, 4.2BSD approach and the STREAMS-based approach to pipes were compatible. POSIX has standardized the programmatic interface to pipes. STREAMS-based pipes have been POSIX compliant for many years and were POSIX compliant in the *SVR4.2* release. The STREAMS-based pipes provided by the *Linux Fast-STREAMS* package provides POSIX compliant STREAMS-based pipes.

As a result, any application utilizing a legacy Linux pipe in a POSIX compliant manner will also be compatible with STREAMS-based pipes.[4]

1.3 Linux Fast-STREAMS

The first STREAMS package for Linux that provided SVR4 STREAMS capabilities was the *Linux STREAMS (LiS)* package originally available from GCOM. This package exhibited incompatibilities with SVR 4.2 STREAMS and other STREAMS implementations, was buggy and performed very poorly on Linux. These difficulties prompted the OpenSS7 Project [SS7] to implement an SVR 4.2 STREAMS package from scratch, with the objective of production quality and high-performance, named *Linux Fast-STREAMS*.

The OpenSS7 Project [SS7] also maintains public and internal versions of the *LiS* package. The last public release was *LiS-2.18.3*; the current internal release version is *LiS-2.18.6*. The current production public release of *Linux Fast-STREAMS* is *streams-0.9.3*.

2 Objective

The objective of the current study is to determine whether, for the Linux operating system, the newer STREAMS-based pipe approach is (from the perspective of performance) a viable replacement for the legacy 4.1BSD/SVR3-style pipes provided by Linux. As a side objective, a comparison is also made to STREAMS-based pipes implemented on the deprecated LiS (Linux STREAMS) package. This comparison will demonstrate one reason why *Linux Fast-STREAMS* was written in the first place.

Misconceptions When developing STREAMS, the authors oft times found that there were a number of preconceptions from Linux advocates about both STREAMS and STREAMS-based pipes, as follows:

- STREAMS is slow.
- STREAMS is more flexible, but less efficient [LML].
- STREAMS performs poorly on uniprocessor and even poorer on SMP.
- STREAMS-based pipes are slow.
- STREAMS-based pipes are unnecessarily complex and cumbersome.

For example, the Linux kernel mailing list has this to say about STREAMS:

> **(REG)** STREAMS allow you to "push" filters onto a network stack. The idea is that you can have a very primitive network stream of data, and then "push" a filter ("module") that implements TCP/IP or whatever on top of that. Conceptually, this is very nice, as it allows clean

3. It has been said of Linux that, without STREAMS, it is just another BSD... ...and not a very good one.

4. This compatibility is exemplified by the perftest(8) program which does not distinguish between legacy and STREAMS-based pipes in their implementation or use.

separation of your protocol layers. Unfortunately, implementing STREAMS poses many performance problems. Some Unix STREAMS based server telnet implementations even ran the data up to user space and back down again to a pseudo-tty driver, which is very inefficient.

STREAMS will **never** be available in the standard Linux kernel, it will remain a separate implementation with some add-on kernel support (that come with the STREAMS package). Linus and his networking gurus are unanimous in their decision to keep STREAMS out of the kernel. They have stated several times on the kernel list when this topic comes up that even optional support will not be included.

(REW, quoting Larry McVoy) "It's too bad, I can see why some people think they are cool, but the performance cost - both on uniprocessors and even more so on SMP boxes - is way too high for STREAMS to ever get added to the Linux kernel."

Please stop asking for them, we have agreement amoungst the head guy, the networking guys, and the fringe folks like myself that they aren't going in.

(REG, quoting Dave Grothe, the STREAMS guy) STREAMS is a good framework for implementing complex and/or deep protocol stacks having nothing to do with TCP/IP, such as SNA. It trades some efficiency for flexibility. You may find the Linux STREAMS package (LiS) to be quite useful if you need to port protocol drivers from Solaris or UnixWare, as Caldera did.

The Linux STREAMS (LiS) package is available for download if you want to use STREAMS for Linux. The following site also contains a dissenting view, which supports STREAMS.

It is possible that the proponents of these statements have worked in the past with an improper or under-performing STREAMS implementation (such as *LiS*); however, the current study aims to prove that none of these statements are correct for the STREAMS-based pipes provided by the high-performance *Linux Fast-STREAMS*.

3 Description

The three implementations tested vary in their implementation details. These implementation details are described below.

3.1 Linux Pipes

Linux pipes are implemented using a file-system approach similar to that of 4.1BSD, or that of SVR3 and SVR2 releases, or their common Bell Laboratories predecessors, as illustrated in *Figure 3*. It should be noted that 4.4BSD (and releases after 4.2BSD) implements pipes using Sockets and the networking subsystem [MBKQ97]. Also, note that SVR4 implemented pipes using STREAMS. As such, the Linux pipe implementation is both archaic and deprecated.

Write side processing. In response to a write(2) system call, message bytes are copied from user space to kernel directly into a preallocated buffer. The tail pointer is pushed on the buffer. If the buffer is full at the time of the system call, the calling process blocks, or the system call fails and returns an error number (`EAGAIN` or `EWOULDBLOCK`).

Read side processing. In response to a read(2) system call, message bytes are copied from the preallocated buffer to user space. The head pointer is pushed on the buffer. If the buffer is empty at the time of the system call, the calling process blocks, or the system call fails and returns an error number (`EAGAIN` or `EWOULDBLOCK`).

Buffering. If a writer goes to write and there is no more room left in the buffer for the requested write, the writer blocks or the system call is failed (`EAGAIN`). If a reader goes to read and there are no bytes in the buffer, the reader blocks or the system call is failed (`EAGAIN`). If there are fewer bytes in the buffer than requested by the read operation, the available bytes are returned. No queueing or flow control is performed.

Scheduling. When a writer is blocked or polling for write, the writer is awoken once there is room to write at least 1 byte into the buffer. When a reader is blocked or polling for read, the reader is awoken once there is at least 1 byte in the buffer.

3.2 STREAMS-based Pipes

STREAMS-based pipes are implemented using a specialized STREAMS driver that connects the read and write queues of two *Stream heads* in a twisted pair configuration as illustrated in *Figure 1*. Aside from a few specialized settings particular to pipes, each *Stream head* acts in the same fashion as the *Stream head* for any other STREAMS device or pseudo-device.

Write side processing. In response to a write(2) system call, message bytes are copied from user space into allocated message blocks. Message blocks are passed downstream to the next module (read *Stream head*) in the *Stream*. If flow control is in effect on the write queue at the time of the system call, the calling process blocks, or the system call fails and returns an error number (`EAGAIN`). Also, STREAMS has a write message coalescing feature that allows message blocks to be held temporarily on the write queue awaiting execution of the write queue service procedure (invoked by the STREAMS scheduler) or the occurrence of another write operation.

Read side processing. In response to a read(2) system call, message blocks are removed from the read queue and message bytes copied from kernel to user space. If there are no message blocks in the read queue at the time of the system call, the calling process blocks, or the system call fails and returns and error number (`EAGAIN`). Also, STREAMS has a read notification feature that causes a read notification message (`M_READ`) containing the requested number of bytes to be issued and passed downstream before blocking. STREAMS has an additional read-fill mode feature which causes the read side to attempt to satisfy the entire read request before returning to the user.

Buffering. If a writer goes to write and the write queue is flow controlled, the writer blocks or the system call is failed (`EAGAIN`). If a reader goes to read and there are no message blocks available, the reader blocks or the system call is failed (`EAGAIN`). If there are fewer bytes available in message blocks on the read queue than requested by the read operation, the available bytes are returned. Normal STREAMS queueing and flow control is performed as message blocks are passed along the write side or removed from the read queue.

Scheduling. When a write is blocked or polling for write, the writer is awoken once flow control subsides on the write side. Flow control subsides when the downstream module's queue on the write side falls below its low water mark, the *Stream* is back-enabled, and the write queue service procedure runs. When a reader is blocked or polling for read, the reader is awoken once the read queue service procedure runs. The read queue service procedure is scheduled when the first message block is placed on the read queue after an attempt to remove a message block from the queue failed. The service procedure runs when the STREAMS scheduler runs.

4 Method

To test the performance of STREAMS-based pipes, the *Linux Fast-STREAMS* package was used [LfS]. The *Linux Fast-STREAMS* package builds and install Linux loadable kernel modules and includes the `perftest` program used for testing. For comparison, the *LiS* package [LiS] was used for comparison.

4.1 Test Program

To test the maximum throughput performance of both legacy pipes and STREAMS-based pipes, a test program was written, called `perftest`. The `perftest` program is part of the *Linux Fast-STREAMS* distribution [LfS]. The test program performs the following actions:

1. Opens either a legacy pipe or a STREAMS-based pipe.
2. Forks two child processes: a writer child process and a reader child process.
3. The writer child process closes the reading end of the pipe.
4. The writer child process starts an interval timer.
5. The writer child process begins writing data to the pipe end with the write(2) system call.
6. As the writer child process writes to the pipe end, it tallies the amount of data written. When the interval timer expires, the tally is output and the interval timer restarted.
7. The reader child process closes the writing end of the pipe.
8. The reader child process starts an interval timer.
9. The reader child process begins reading data from the pipe end with the read(2) system call.
10. As the reader child process reads from the pipe end, it tallies the amount of data written. When the interval timer expires, the tally is output and the interval timer restarted.

The test program thus simulates a typical use of pipes in a Linux system. The `perftest_script` performance testing script was used to obtain repeatable results (see *Appendix A*).

4.2 Distrbutions Tested

To remove the dependence of test results on a particular Linux kernel or machine, various Linux distributions were used for testing. The distributions tested are as follows:

Distribution	Kernel
RedHat 7.2	2.4.20-28.7
CentOS 4	2.6.9-5.0.3.EL
CentOS 5	2.6.18-8-el5
SuSE 10.0 OSS	2.6.13-15-default
Ubuntu 6.10	2.6.17-11-generic
Ubuntu 7.04	2.6.20-15-generic
Fedora Core 6	2.6.20-1.2933.fc6

4.3 Test Machines

To remove the dependence of test results on a particular machine, various machines were used for testing as follows:

Hostname	Processor	Memory	Architecture
porky	2.57GHz PIV	1Gb (333MHz)	i686 UP
pumbah	2.57GHz PIV	1Gb (333MHz)	i686 UP
daisy	3.0GHz i630 HT	1Gb (400MHz)	x86_64 SMP
mspiggy	1.7GHz PIV	1Gb (333MHz)	i686 UP

5 Results

The results for the various distributions and machines is tabulated in *Appendix B*. The data is tabulated as follows:

Performance. Performance is charted by graphing the number of writes per second against the logarithm of the write size.

Delay. Delay is charted by graphing the number of second per write against the write size. The delay can be modelled as a fixed write overhead per write operation and a fixed overhead per byte written. This model results in a linear graph with the intercept at 1 byte per write representing the fixed per-write overhead, and the slope of the line representing the per-byte cost. As all implementations use the same primary mechanisms for copying bytes to and from user space, it is expected that the slope of each graph will be similar and that the intercept will reflect most implementation differences.

Throughput. Throughput is charted by graphing the logarithm of the product of the number of writes per second and the message size against the logarithm of the message size. It is expected that these graphs will exhibit strong log-log-linear (power function) characteristics. Any curvature in these graphs represent throughput saturation.

Improvement. Improvement is charted by graphing the quotient of the writes per second of the implementation and the writes per second of the Linux legacy pipe implementation as a percentage against the write size. Values over 0% represent an improvement over Linux legacy pipes, whereas values under 0% represent the lack of an improvement.

The results are organized in the section that follow in order of the machine tested.

5.1 Porky

Porky is a 2.57GHz Pentium IV (i686) uniprocessor machine with 1Gb of memory. Linux distributions tested on this machine are as follows:

Distribution	Kernel
Fedora Core 6	2.6.20-1.2933.fc6
CentOS 4	2.6.9-5.0.3.EL
SuSE 10.0 OSS	2.6.13-15-default
Ubuntu 6.10	2.6.17-11-generic
Ubuntu 7.04	2.6.20-15-generic

5.1.1 Fedora Core 6

Fedora Core 6 is the most recent full release Fedora distribution. This distribution sports a 2.6.20-1.2933.fc6 kernel with the latest patches. This is the `x86` distribution with recent updates.

Performance. *Figure 4* illustrates the performance of *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be see from *Figure 4*, the performance of *LiS* is dismal across the entire range of write sizes. The performance of *Linux Fast-STREAMS* STREAMS-based pipes, on the other hand, is superior across the entire range of write sizes. Performance of *Linux Fast-STREAMS* is a full order of magnitude better than *LiS*.

Delay. *Figure 5* illustrates the average write delay for *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. The slope of all three curves is comparable and about the same. This indicates that each implementation is only slightly dependent upon the size of the message and each implementation has a low per-byte processing overhead. This is as expected as pipes primarily copy data from user space to the kernel just to copy it back to user space on the other end. Note that the intercepts, on the other hand, differ to a significant extent. *Linux Fast-STREAMS* STREAMS-based pipes have by far the lowest per-write overhead (about half that of the Linux legacy pipes, and a sixth of *LiS* pipes).

Throughput. *Figure 6* illustrates the throughput experienced by *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 6*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation (regardless of performance).

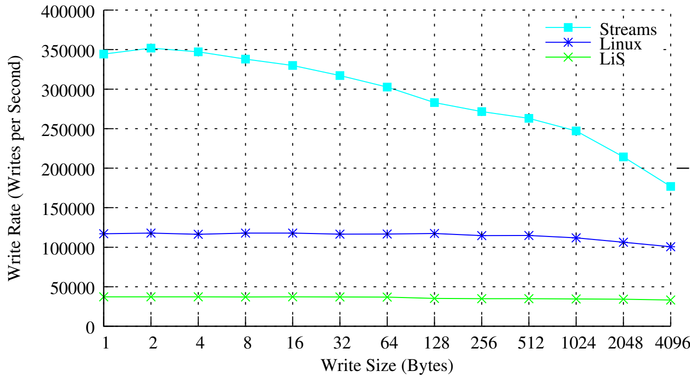

Figure 4: FC6 on Porky Performance

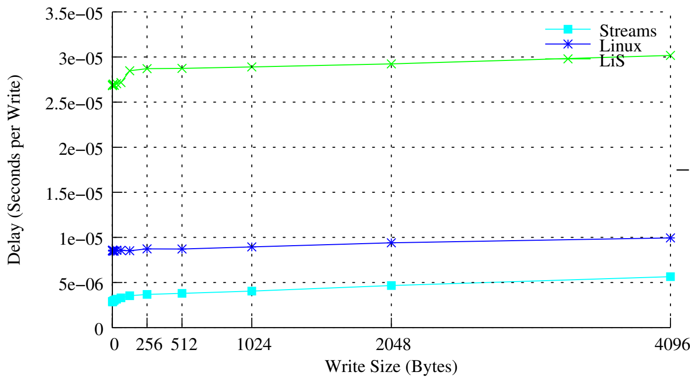

Figure 5: FC6 on Porky Delay

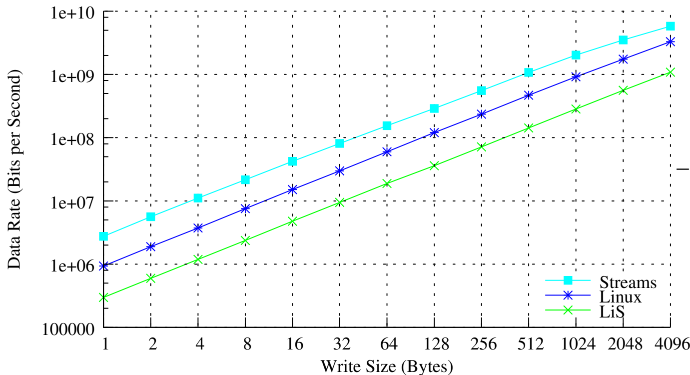

Figure 6: FC6 on Porky Throughput

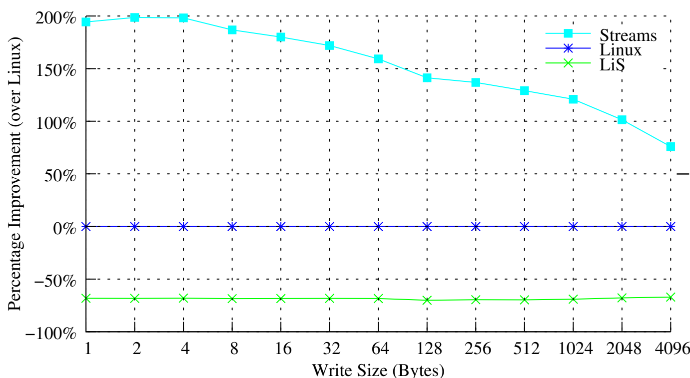

Figure 7: FC6 on Porky Comparison

Improvement. *Figure 7* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is marked: improvements range from a significant 75% increase in performance at large write sizes, to a staggering 200% increase in performance at lower write sizes.

5.1.2 CentOS 4.0

CentOS 4.0 is a clone of the RedHat Enterprise 4 distribution. This is the `x86` version of the distribution. The distribution sports a 2.6.9-5.0.3.EL kernel.

Performance. *Figure 8* illustrates the performance of *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be see from *Figure 8*, the performance of *LiS* is dismal across the entire range of write sizes. The performance of *Linux Fast-STREAMS* STREAMS-based pipes, on the other hand, is superior across the entire range of write sizes. Performance of *Linux Fast-STREAMS* is a full order of magnitude better than *LiS*.

Delay. *Figure 9* illustrates the average write delay for *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. The slope of all three curves is comparable and about the same. This indicates that each implementation is only slightly dependent upon the size of the message and each implementation has a low per-byte processing overhead. This is as expected as pipes primarily copy data from user space to the kernel just to copy it back to user space on the other end. Note that the intercepts, on the other hand, differ to a significant extent. *Linux Fast-STREAMS* STREAMS-based pipes have by far the lowest per-write overhead (about half that of the Linux legacy pipes, and a sixth of *LiS* pipes).

Throughput. *Figure 10* illustrates the throughput experienced by *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 10*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation (regardless of performance).

Improvement. *Figure 11* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is marked: improvements range from a significant 100% increase in performance at large write sizes, to a staggering 275% increase in performance at lower write sizes.

5.1.3 SuSE 10.0 OSS

SuSE 10.0 OSS is the public release version of the SuSE/Novell distribution. There have been two releases subsequent to this one: the 10.1 and recent 10.2 releases. The SuSE 10 release sports a 2.6.13 kernel and the 2.6.13-15-default kernel was the tested kernel.

Performance. *Figure 12* illustrates the performance of *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be see from *Figure 12*, the performance of *LiS* is dismal across the entire range of write sizes. The performance of *Linux Fast-STREAMS* STREAMS-based pipes, on the other hand, is superior across the entire range of write sizes. Performance of *Linux Fast-STREAMS* is a full order of magnitude better than *LiS*.

Delay. *Figure 13* illustrates the average write delay for *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a

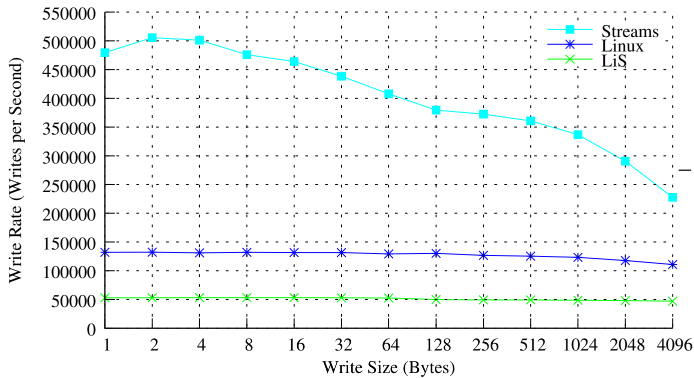

Figure 8: CentOS 4.0 on Porky Performance

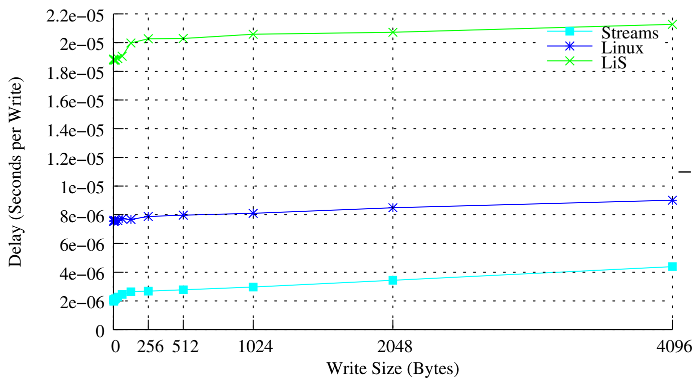

Figure 9: CentOS 4.0 on Porky Delay

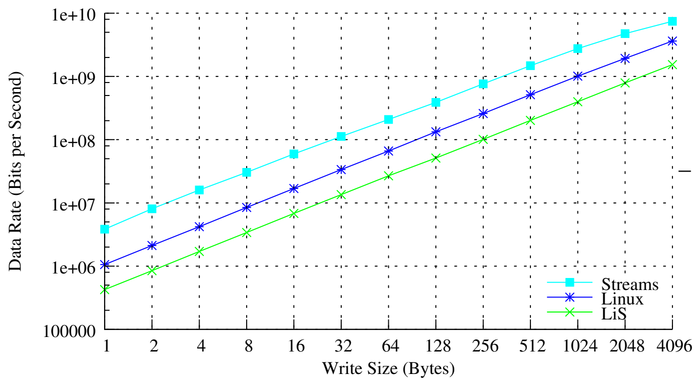

Figure 10: CentOS 4.0 on Porky Throughput

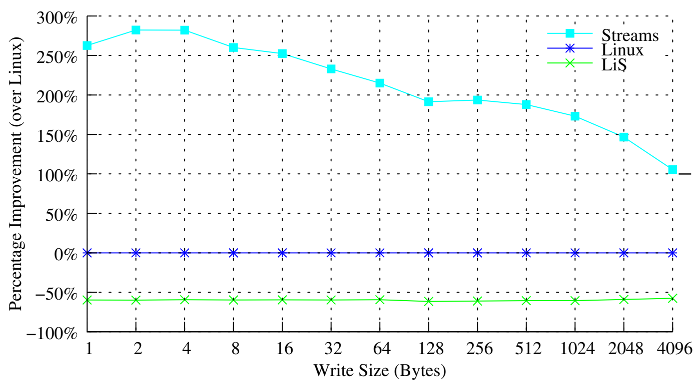

Figure 11: CentOS 4.0 on Porky Comparison

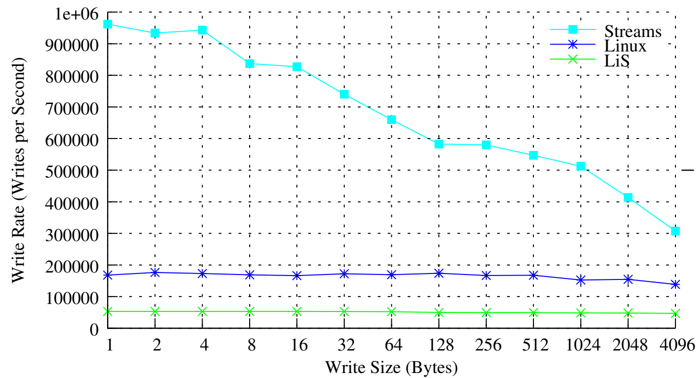

Figure 12: SuSE 10.0 OSS on Porky Performance

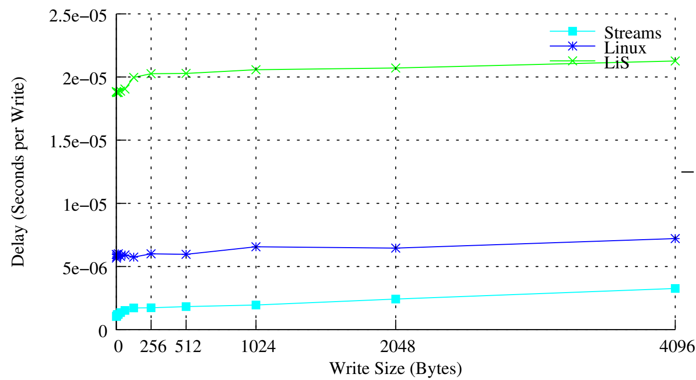

Figure 13: SuSE 10.0 OSS on Porky Delay

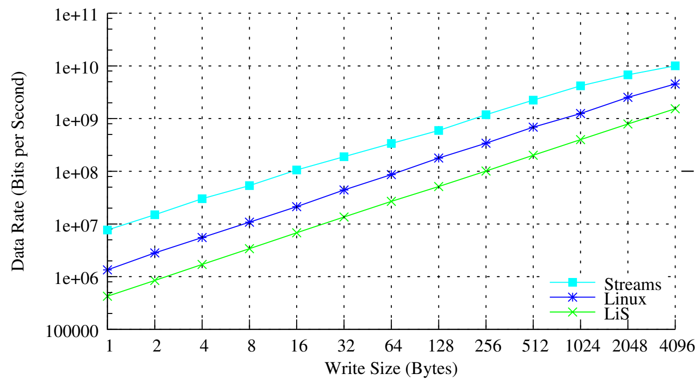

Figure 14: SuSE 10.0 OSS on Porky Throughput

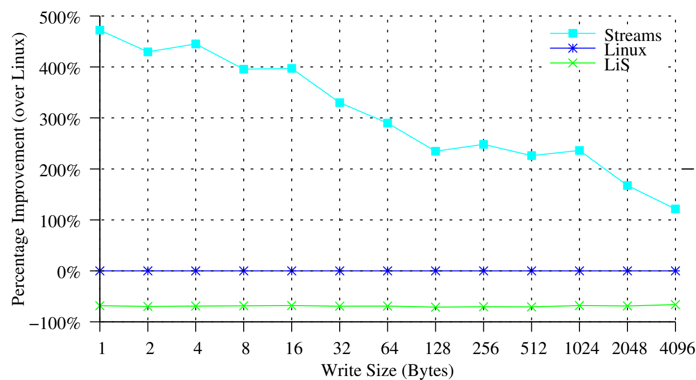

Figure 15: SuSE 10.0 OSS on Porky Comparison

range of write sizes. The slope of the delay curves are similar for all implementations, as expected. The zero intercept of *Linux Fast-STREAMS* is, however, far superior to that of legacy Linux and a full order of magnitude better than the under-performing *LiS*.

Throughput. *Figure 14* illustrates the throughput experienced by *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 14*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation. The *Linux Fast-STREAMS* curve exhibits a downward concave characteristic at large message sizes indicating that the memory bus saturates at about 10Gbps.

Improvement. *Figure 15* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is significant: improvements range from a 100% increase in performance at large write sizes, to a 475% increase in performance at lower write sizes.

5.1.4 Ubuntu 6.10

Ubuntu 6.10 is the current release of the Ubuntu distribution. The Ubuntu 6.10 release sports a 2.6.15 kernel. The tested distribution had current updates applied.

Performance. *Figure 20* illustrates the performance of *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be see from *Figure 20*, the performance of *LiS* is dismal across the entire range of write sizes. The performance of *Linux Fast-STREAMS* STREAMS-based pipes, on the other hand, is superior across the entire range of write sizes. Performance of *Linux Fast-STREAMS* is a full order of magnitude better than *LiS*.

Delay. *Figure 21* illustrates the average write delay for *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. Again, the slope of the delay curves is similar, but *Linux Fast-STREAMS* exhibits a greatly reduced intercept indicating superior per-message overheads.

Throughput. *Figure 22* illustrates the throughput experienced by *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 22*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation. Again *Linux Fast-STREAMS* appears to saturate the memory bus approaching 10Gbps.

Improvement. *Figure 23* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is significant: improvements range from a 75% increase in performance at large write sizes, to a 200% increase in performance at lower write sizes.

5.1.5 Ubuntu 7.04

Ubuntu 7.04 is the current release of the Ubuntu distribution. The Ubuntu 7.04 release sports a 2.6.20 kernel. The tested distribution had current updates applied.

Performance. *Figure 20* illustrates the performance of *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be see from *Figure 20*, the performance of *LiS* is dismal across the entire range of write sizes. The performance of *Linux Fast-STREAMS* STREAMS-based pipes, on the other hand, is superior across the entire range of write sizes. Performance of *Linux Fast-STREAMS* is a full order of magnitude better than *LiS*.

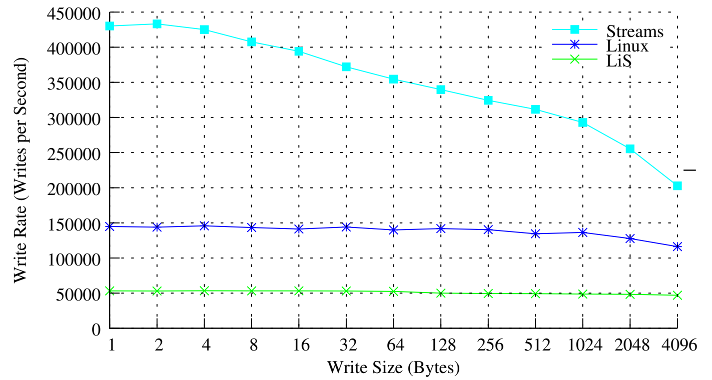

Figure 16: Ubuntu 6.10 on Porky Performance

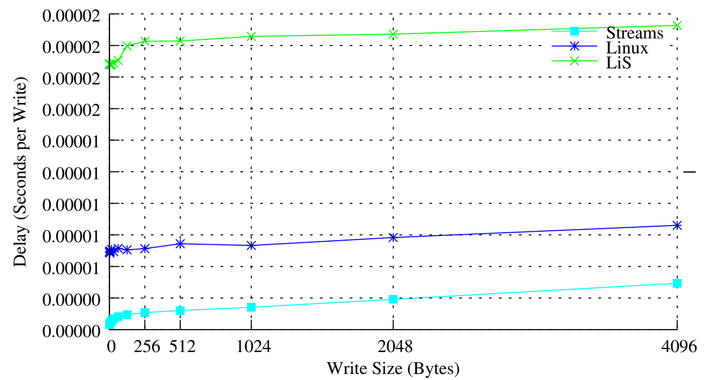

Figure 17: Ubuntu 6.10 on Porky Delay

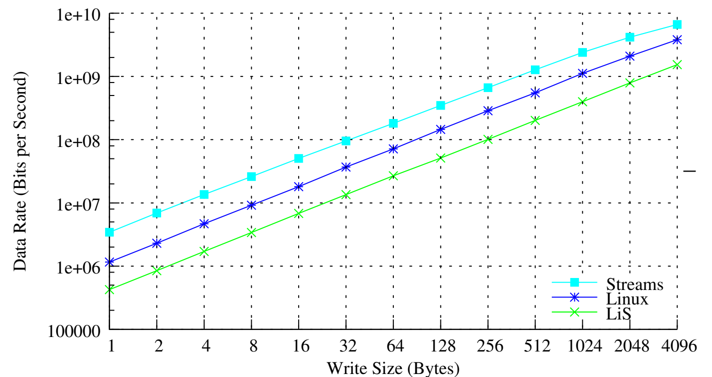

Figure 18: Ubuntu 6.10 on Porky Throughput

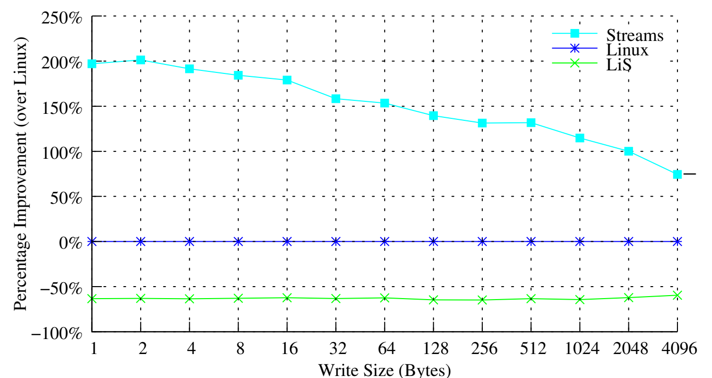

Figure 19: Ubuntu 6.10 on Porky Comparison

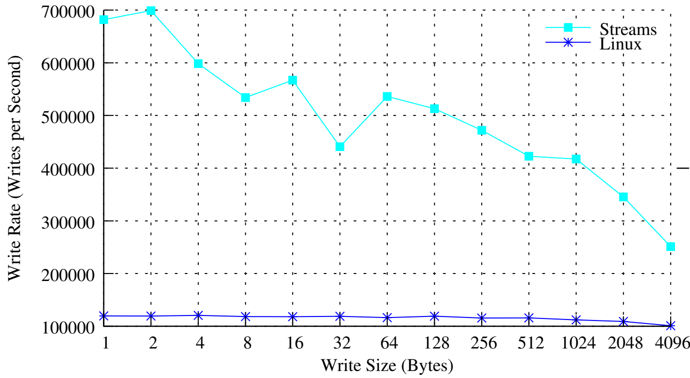

Figure 20: Ubuntu 7.04 on Porky Performance

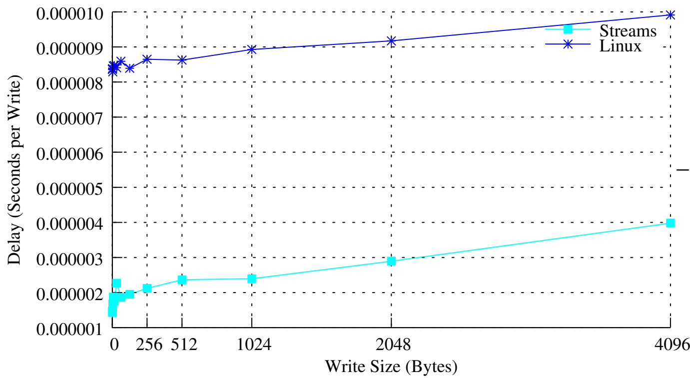

Figure 21: Ubuntu 7.04 on Porky Delay

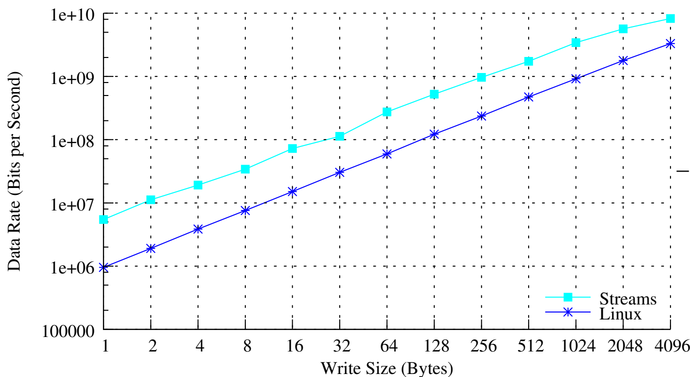

Figure 22: Ubuntu 7.04 on Porky Throughput

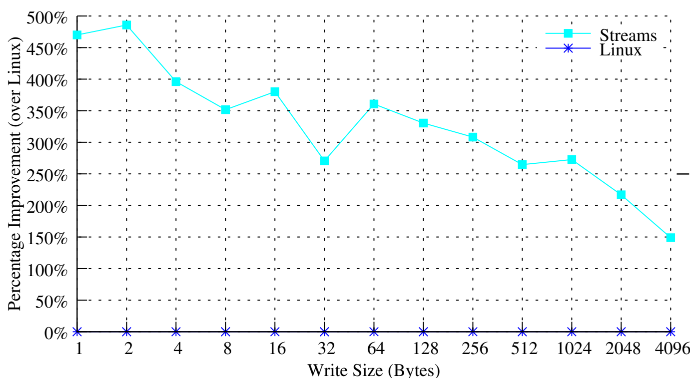

Figure 23: Ubuntu 7.04 on Porky Comparison

Delay. *Figure 21* illustrates the average write delay for *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. Again, the slope of the delay curves is similar, but *Linux Fast-STREAMS* exhibits a greatly reduced intercept indicating superior per-message overheads.

Throughput. *Figure 22* illustrates the throughput experienced by *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 22*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation. Again *Linux Fast-STREAMS* appears to saturate the memory bus approaching 10Gbps.

Improvement. *Figure 23* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is significant: improvements range from a 75% increase in performance at large write sizes, to a 200% increase in performance at lower write sizes.

5.2 Pumbah

Pumbah is a 2.57GHz Pentium IV (i686) uniprocessor machine with 1Gb of memory. This machine differs from Porky in memory type only (Pumbah has somewhat faster memory than Porky.) Linux distributions tested on this machine are as follows:

Distribution	Kernel
RedHat 7.2	2.4.20-28.7

Pumbah is a control machine and is used to rule out differences between recent 2.6 kernels and one of the oldest and most stable 2.4 kernels.

5.2.1 RedHat 7.2

RedHat 7.2 is one of the oldest (and arguably the most stable) glibc2 based releases of the RedHat distribution. This distribution sports a 2.4.20-28.7 kernel. The distribution has all available updates applied.

Performance. *Figure 24* illustrates the performance of *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be see from *Figure 24*, the performance of *LiS* is dismal across the entire range of write sizes. The performance of *Linux Fast-STREAMS* STREAMS-based pipes, on the other hand, is superior across the entire range of write sizes. At a write size of one byte, the performance of *Linux Fast-STREAMS* is an order of magnitude greater than *LiS*.

Delay. *Figure 25* illustrates the average write delay for *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. The slope of all three graphs is similar, indicating that memory caching and copy to and from user performance on a byte-by-byte basis is similar. The intercepts, on the other hand, are drastically different. *LiS* per-message overheads are massive. *Linux Fast-STREAMS* and Linux legacy pipes are far better. STREAMS-based pipes have about one third of the per-message overhead of legacy pipes.

Throughput. *Figure 26* illustrates the throughput experienced by *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 26*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation (despite performance differences). On Pumbah, as was experienced on Porky, *Linux Fast-STREAMS* is beginning to saturate the memory bus at 10Gbps.

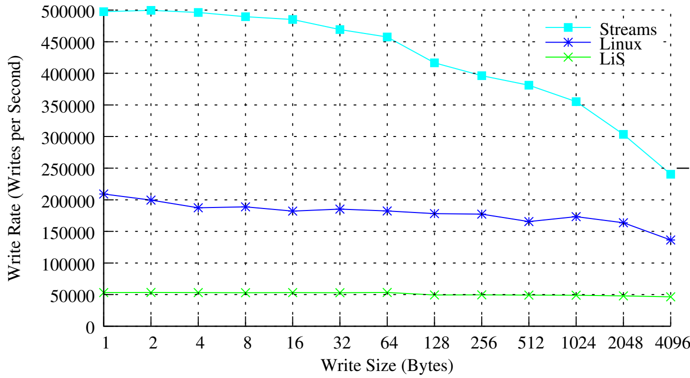

Figure 24: RH7.2 on Pumbah Performance

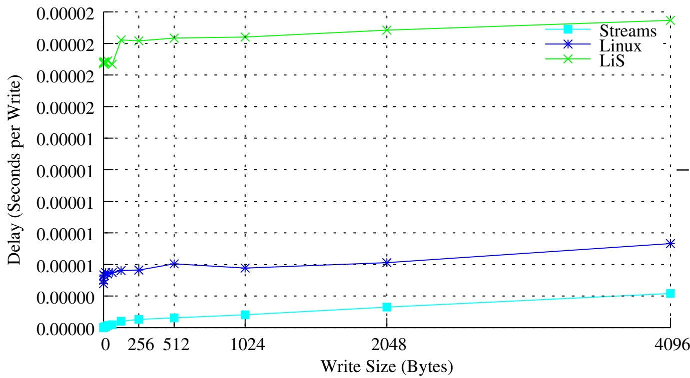

Figure 25: RH7.2 on Pumbah Delay

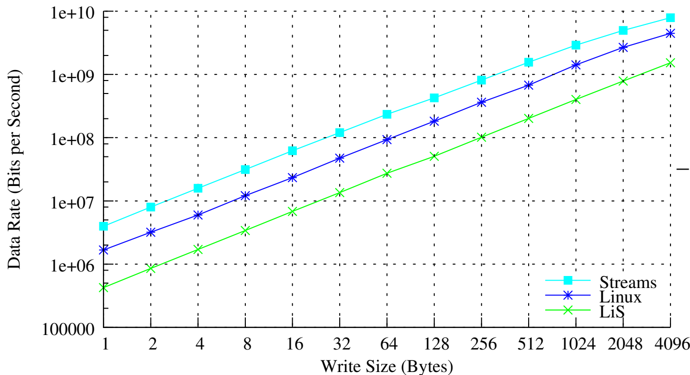

Figure 26: RH7.2 on Pumbah Throughput

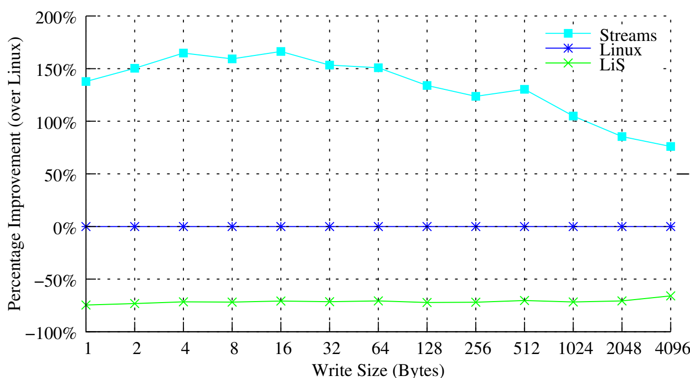

Figure 27: RH7.2 on Pumbah Comparison

Improvement. *Figure 27* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is significant: improvements range from a 75% increase in performance at large write sizes, to a 175% increase in performance at lower write sizes. *LiS* pipes waddle in at a 75% *decrease* in performance.

5.3 Daisy

Daisy is a 3.0GHz i630 (x86_64) hyper-threaded machine with 1Gb of memory. Linux distributions tested on this machine are as follows:

Distribution	Kernel
Fedora Core 6	2.6.20-1.2933.fc6
CentOS 5	2.6.18-8-el5
CentOS 5.2	2.6.18-92.1.6.el5.centos.plus

This machine is used as an SMP control machine. Most of the test were performed on uniprocessor non-hyper-threaded machines. This machine is hyper-threaded and runs full SMP kernels. This machine also supports EMT64 and runs `x86_64` kernels. It is used to rule out both SMP differences as well as 64-bit architecture differences.

5.3.1 Fedora Core 6 (x86_64)

Fedora Core 6 is the most recent full release Fedora distribution. This distribution sports a 2.6.20-1.2933.fc6 kernel with the latest patches. This is the `x86_64` distribution with recent updates.

Performance. *Figure 28* illustrates the performance of *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be see from *Figure 28*, the performance of *LiS* is dismal across the entire range of write sizes. The performance of *Linux Fast-STREAMS* STREAMS-based pipes, on the other hand, is superior across the entire range of write sizes. The performance of *Linux Fast-STREAMS* is almost an order of magnitude greater than that of *LiS*.

Delay. *Figure 29* illustrates the average write delay for *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. Again the slope appears to be the same for all implementations, except Linux legacy pipes which exhibit some anomalies below 1024 byte write sizes. The intercept for *Linux Fast-STREAMS* is again much superior to the other two implementations.

Throughput. *Figure 30* illustrates the throughput experienced by *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 30*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation.

Improvement. *Figure 31* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is significant: improvements range from a 100% increase in performance at large write sizes, to a 175% increase in performance at lower write sizes. *LiS* again drags in at -75%.

5.3.2 CentOS 5 (x86_64)

CentOS 5 is the most recent full release CentOS distribution. This distribution sports a 2.6.18-8-el5 kernel with the latest patches. This is the `x86_64` distribution with recent updates.

Performance. *Figure 32* illustrates the performance of *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a

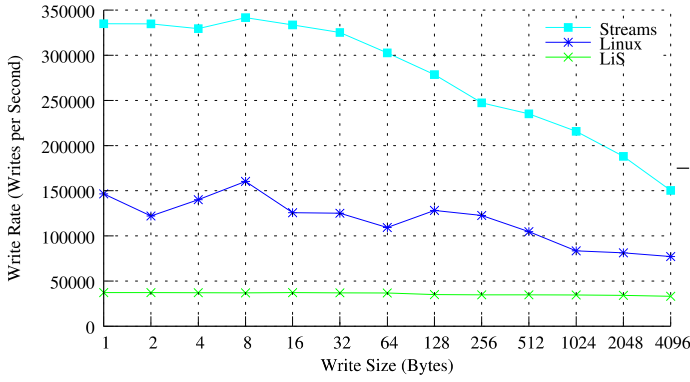

Figure 28: FC6 on Daisy Performance

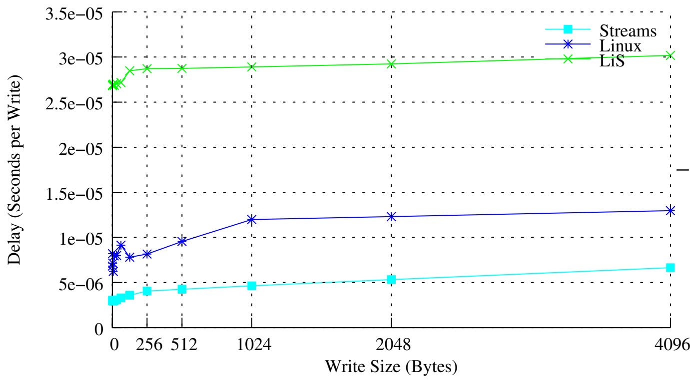

Figure 29: FC6 on Daisy Delay

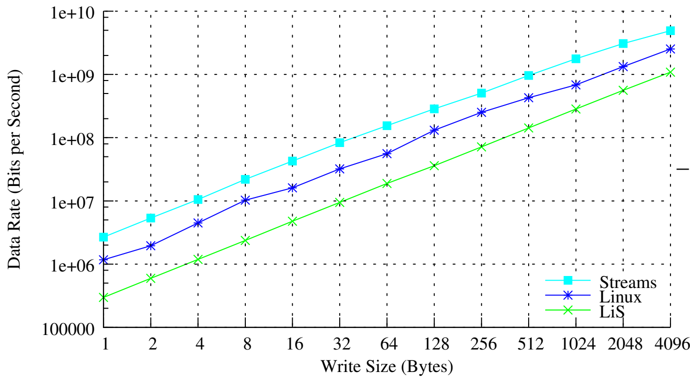

Figure 30: FC6 on Daisy Throughput

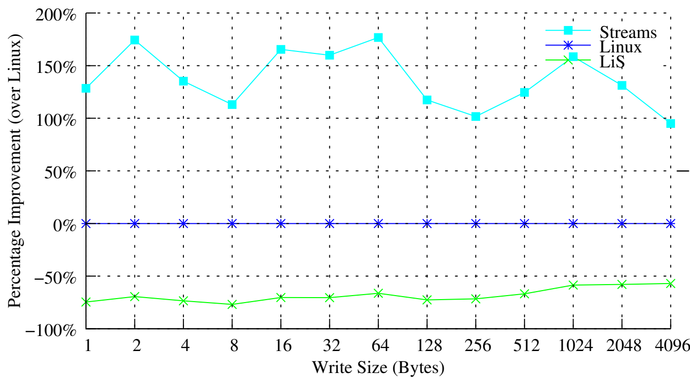

Figure 31: FC6 on Daisy Comparison

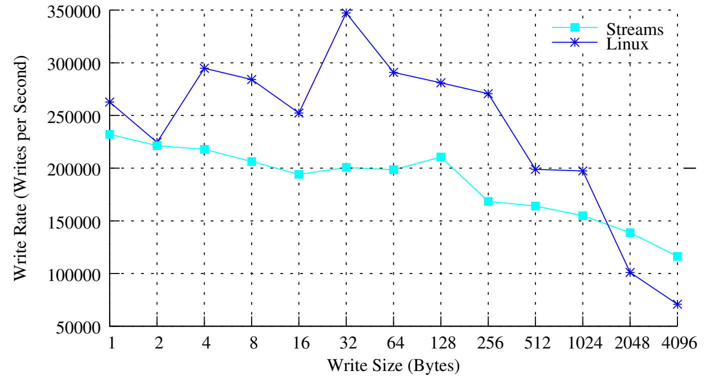

Figure 32: CentOS 5 on Daisy Performance

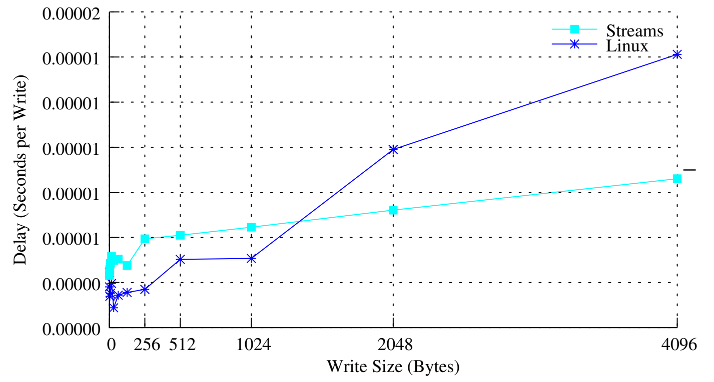

Figure 33: CentOS 5 on Daisy Delay

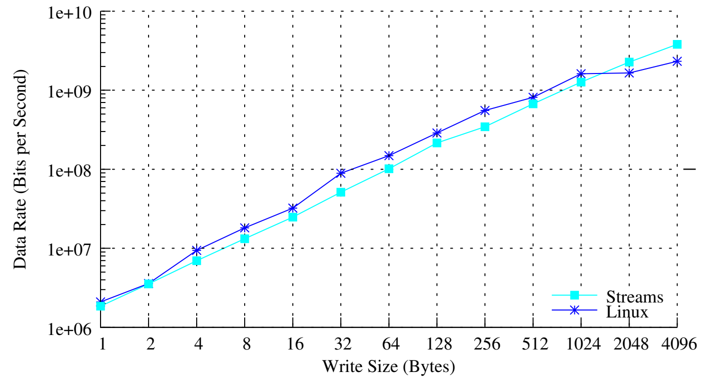

Figure 34: CentOS 5 on Daisy Throughput

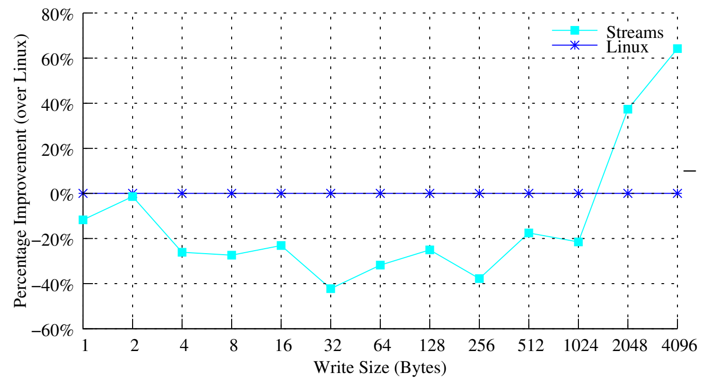

Figure 35: CentOS 5 on Daisy Comparison

range of write sizes. As can be see from *Figure 32*, the performance of *LiS* is dismal across the entire range of write sizes. The performance of *Linux Fast-STREAMS* STREAMS-based pipes, on the other hand, is superior across the entire range of write sizes. The performance of *Linux Fast-STREAMS* is almost an order of magnitude greater than that of *LiS*.

Delay. *Figure 33* illustrates the average write delay for *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. Again the slope appears to be the same for all implementations, except Linux legacy pipes which exhibit some anomalies below 1024 byte write sizes. The intercept for *Linux Fast-STREAMS* is again much superior to the other two implementations.

Throughput. *Figure 34* illustrates the throughput experienced by *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 34*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation.

Improvement. *Figure 35* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is significant: improvements range from a 100% increase in performance at large write sizes, to a 175% increase in performance at lower write sizes. *LiS* again drags in at -75%.

5.3.3 CentOS 5.2 (x86_64)

CentOS 5.2 is the most recent full release CentOS distribution. This distribute sports a 2.6.18-92.1.6.el5.centos.plus kernel with the latest patches. This is the `x86_64` distribution with recent updates.

This is a test result set that was updated July 26, 2008. The additional options, `-H`, `-M`, `-F` and `-w` were added to the `perftest_script` command line. Also, `streams-0.9.2.4` was tested.

Performance. *Figure 36* illustrates the performance of *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be see from *Figure 36*, the performance of *Linux Fast-STREAMS* STREAMS-based pipes is superior across the entire range of write sizes. The performance of *Linux Fast-STREAMS* is significantly greater (by a factor of 4 through 7) than Linux legacy pipes at smaller write sizes.

The performance boost experienced by *Linux Fast-STREAMS* at write sizes beneath 128 is primariliy due to the write coallescing feature (hold feature) of the Stream head combined with the fact that the fast-buffer sizes for `x86_64` is 128 bytes. The performance boost experienced across the entire range is primarily due to the read-fill option combined with full-sized reads.

Note that it was not possible to get *LiS* running on this kernel.

Delay. *Figure 37* illustrates the average write delay for *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. Again the slope appears to be similar for both implementations if a little bit erratic. The intercept for *Linux Fast-STREAMS* is again much superior than *Linux* legacy pipes.

Again, the delay drop experienced by *Linux Fast-STREAMS* at write sizes beneath 128 is primariliy due to the write coallescing feature (hold feature) of the Stream head combined with the fact that the fast-buffer sizes for `x86_64` is 128 bytes. The delay drop experienced across the entire range is

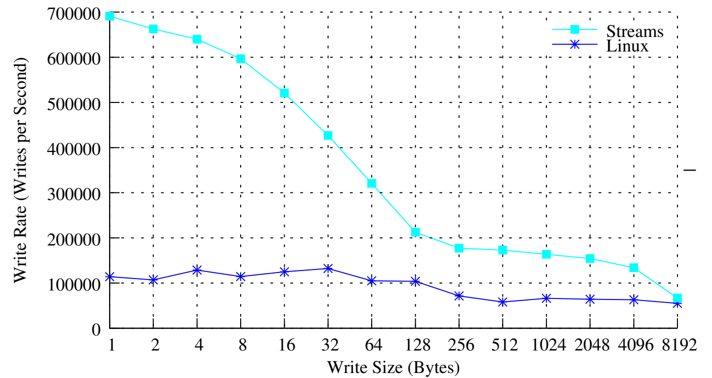

Figure 36: CentOS 5.2 on Daisy Performance

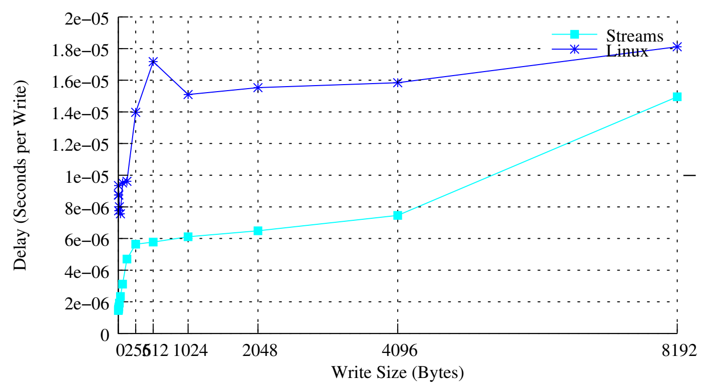

Figure 37: CentOS 5.2 on Daisy Delay

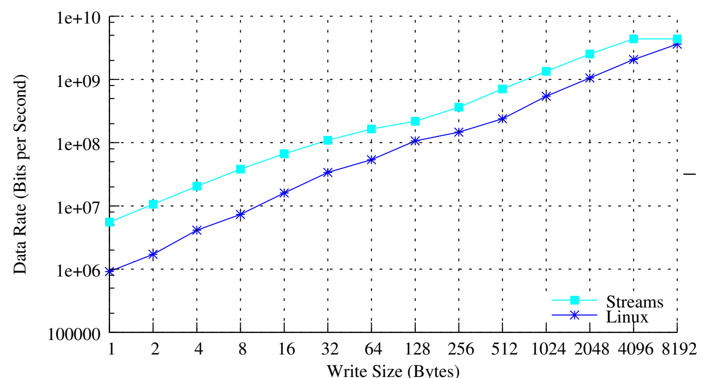

Figure 38: CentOS 5.2 on Daisy Throughput

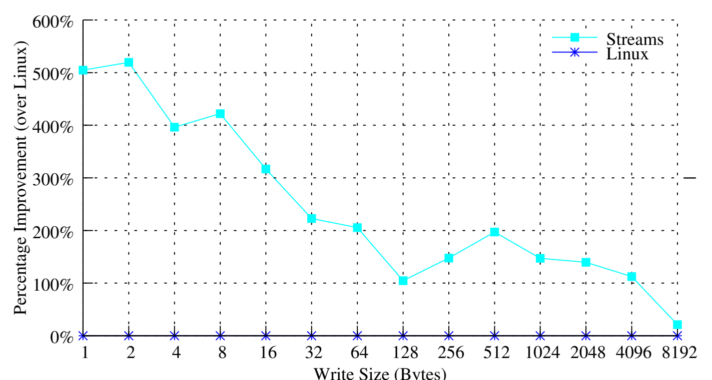

Figure 39: CentOS 5.2 on Daisy Comparison

primarily due to the read-fill option combined with full-sized reads.

Note that it was not possible to get *LiS* running on this kernel.

Throughput. *Figure 38* illustrates the throughput experienced by *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 38*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation.

Again, the throughput increase experienced by *Linux Fast-STREAMS* at write sizes beneath 128 is primariliy due to the write coallescing feature (hold feature) of the Stream head combined with the fact that the fast-buffer sizes for `x86_64` is 128 bytes. The throughput increase experienced across the entire range is primarily due to the read-fill option combined with full-sized reads.

Note that it was not possible to get *LiS* running on this kernel.

Improvement. *Figure 39* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is significant: improvements range from a 100% increase in performance at large write sizes, to a staggering 500% increase in performance at lower write sizes.

Again, the improvements experienced by *Linux Fast-STREAMS* at write sizes beneath 128 is primariliy due to the write coallescing feature (hold feature) of the Stream head combined with the fact that the fast-buffer sizes for `x86_64` is 128 bytes. The improvements experienced across the entire range is primarily due to the read-fill option combined with full-sized reads.

Note that it was not possible to get *LiS* running on this kernel.

5.4 Mspiggy

Mspiggy is a 1.7Ghz Pentium IV (M-processor) uniprocessor notebook (Toshiba Satellite 5100) with 1Gb of memory. Linux distributions tested on this machine are as follows:

Distribution	Kernel
SuSE 10.0 OSS	2.6.13-15-default

Note that this is the same distribution that was also tested on Porky. The purpose of testing on this notebook is to rule out the differences between machine architectures on the test results. Tests performed on this machine are control tests.

5.4.1 SuSE 10.0 OSS

SuSE 10.0 OSS is the public release version of the SuSE/Novell distribution. There have been two releases subsequent to this one: the 10.1 and recent 10.2 releases. The SuSE 10 release sports a 2.6.13 kernel and the 2.6.13-15-default kernel was the tested kernel.

Performance. *Figure 40* illustrates the performance of *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be see from *Figure 40*, the performance of *LiS* is dismal across the entire range of write sizes. The performance of *Linux Fast-STREAMS* STREAMS-based pipes, on the other hand, is superior across the entire range of write sizes. *Linux Fast-STREAMS* again performs a full order of magnitude better than *LiS*.

Delay. *Figure 41* illustrates the average write delay for *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a

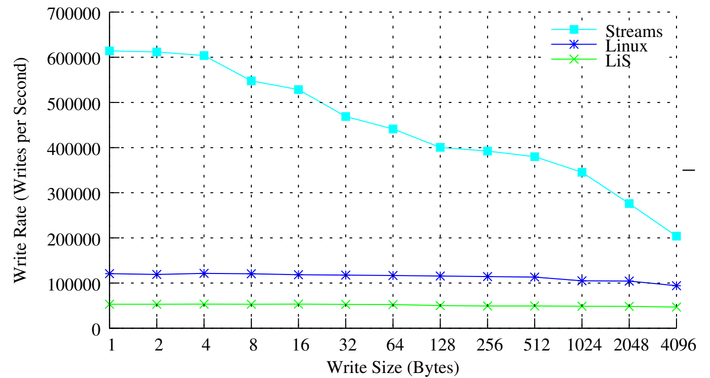

Figure 40: SuSE 10.0 OSS on Mspiggy Performance

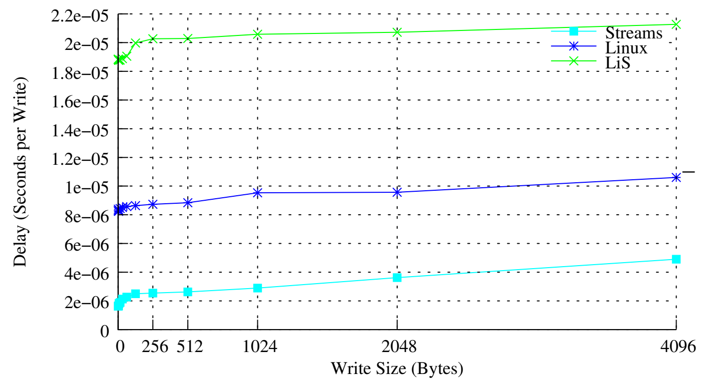

Figure 41: SuSE 10.0 OSS on Mspiggy Delay

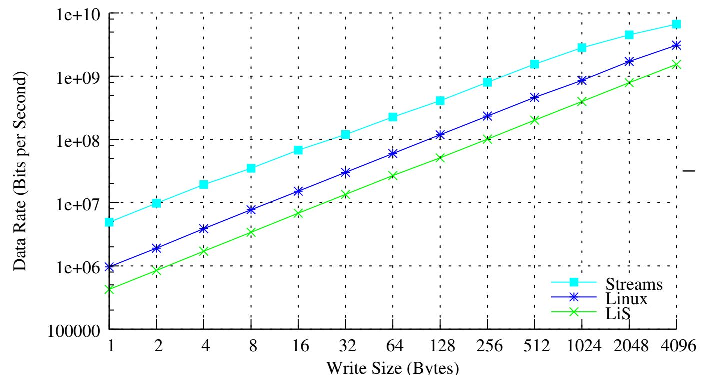

Figure 42: SuSE 10.0 OSS on Mspiggy Throughput

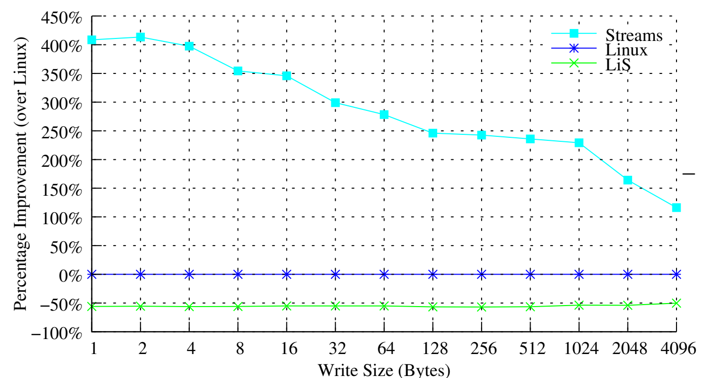

Figure 43: SuSE 10.0 OSS on Mspiggy Comparison

range of write sizes. The slope of the delay curves is, again, similar, but the intercept for *Linux Fast-STREAMS* is far superior.

Throughput. *Figure 42* illustrates the throughput experienced by *LiS*, *Linux Fast-STREAMS* and Linux legacy pipes across a range of write sizes. As can be seen from *Figure 42*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation. *Linux Fast-STREAMS* again begins to saturate the memory bus at 10Gbps.

Improvement. *Figure 43* illustrates the improvement over Linux legacy pipes of *Linux Fast-STREAMS* STREAMS-based pipes. The improvement of *Linux Fast-STREAMS* over Linux legacy pipes is significant: improvements range from a 100% increase in performance at large write sizes, to a staggering 400% increase in performance at lower write sizes.

6 Analysis

The results across the various distributions and machines tested are consistent enough to draw some conclusions from the test results.

6.1 Discussion

The test results reveal that the maximum throughput performance, as tested by the `perftest` program, of STREAMS-based pipes (as implemented by *Linux Fast-STREAMS*) is remarkably superior to that of legacy Linux pipes, regardless of write or read sizes. In fact, STREAMS-based pipe performance at smaller write/read sizes is significantly greater (as much as 200-400%) than that of legacy pipes. The performance of *LiS* is dismal (approx. 75% *decrease*) compared to legacy Linux pipes.

Looking at only the legacy Linux and *Linux Fast-STREAMS* implementations, the difference can be described by analyzing the implementations.

Write side processing. Linux legacy pipes use a simple method on the write side of the pipe. The pipe copies bytes from the user into a preallocated page, by pushing a tail pointer. If there is a sleeping reader process, the process is awoken. If there is no more room in the buffer, the write process sleeps or fails.

STREAMS, on the other hand, uses full flow control. On the write side of the STREAMS-based pipe, the *Stream head* allocates a message block and copies the bytes from the user to the message block and places the message block onto the *Stream*. This results in placing the message on the opposite *Stream head*. If a reader is sleeping on the opposite *Stream head*, the *Stream head*'s read queue service procedure is scheduled. If the *Stream* is flow controlled, the writing process sleeps or fails.

STREAMS has the feature that when a reader finds insufficient bytes available to satisfy the read, it issues an `M_READ` message downstream requesting a specific number of bytes. When the writing *Stream head* receives this message, it attempts to satisfy the full read request before sending data downstream.

Linux Fast-STREAMS also has the feature that when flow control is exerted, it saves the message buffer and a subsequent write of the same size is added to the same buffer.

Read side processing. On the read side of the legacy pipe, bytes are copied from the preallocated page buffer to the user, pulling a head pointer. If there are no bytes available to be read in the buffer, the reading process sleeps or fails. When bytes have been read from the buffer and a process is sleeping waiting to write, the sleeping process is awoken.

STREAMS again uses full flow control. On the read side of the STREAMS-based pipe, messages are removed from the *Stream head* read queue, copied to the user, and then the message is either freed (when all the bytes contained are consumed) or placed back on the *Stream head* read queue. If the read queue was previously full and falls beneath the low water mark for the read queue, the *Stream* is back-enabled. Back-enabling results in the service procedure of the write side queue of the other *Stream head* to be scheduled for service. If there are no bytes available to be read, the reading process sleeps or fails.

STREAMS has the additional feature that if there are no bytes to be read, it can issue an `M_READ` message downstream requesting the number of bytes that were issued to the read(2) system call.

Buffering. There are two primary differences in the buffering approaches used by legacy and STREAMS-based pipes:

1. Legacy pipes use preallocated pinned kernel pages to store data using a simply head and tail pointer approach.
2. STREAMS-based pipes use full flow control with STREAMS message blocks and message queues.

One would expect that the STREAMS-based approach would present significant overheads in comparison to the legacy approach; however, the lack of flow control in the Linux approach is problematic.

Scheduling. Legacy pipes schedule by waking a reading process whenever data is available in the buffer to be read, and waking a writing process whenever there is room available in the buffer to write. While accomplishing buffering, this does not provide flow control or scheduling. By not providing even the hysteresis afforded by Sockets, the write and read side thrash the scheduler as bytes are written to and removed from the pipe.

STREAMS-based pipes, on the other hand, use the scheduling mechanisms of STREAMS. When messages are written to the reading *Stream head* and a reader is sleeping, the service procedure for the reading *Stream head*'s read queue is scheduled for later execution. When the STREAMS scheduler later runs, the reading process is awoken. When message are read from the reading *Stream head* read queue and the queue was previously flow controlled, and the byte count falls below the low water mark defined for the queue, the writing *Stream head* write queue service procedure is scheduled. Once the STREAMS scheduler later runs, the writing process is awoken.

Linux Fast-STREAMS is designed to run tasks queued to the STREAMS scheduler on the same processor as the queueing process or task. This avoids unnecessary context switches.

The STREAMS-based pipe approach results in fewer wakeup events being generated. Because there are fewer wakeup events, there are fewer context switches. The reading process is permitted to consume more messages before the writing process is awoken; and the writing process is permitted to write more messages before the reading process is awoken.

Result. The result of the differences between the legacy and the STREAMS based approach is that fewer context switches result: writing processes are allowed to write more messages before a blocked reader is awoken and the reading process is allowed to read more messages before a blocked writer is awoken. This results in greater code path and data cache efficiency and significantly less scheduler thrashing between the reading and writing process.

The increased performance of the STREAMS-based pipes can be explained as follows:

- The STREAMS message coalescing features allows the complexity of the write side process to approach that of the

legacy approach. This feature provides a boost to performance at message sizes smaller than a `FASTBUF`. The size of a `FASTBUF` on 32-bit systems is 64 bytes; on 64-bit systems, 128 bytes. (However, this STREAMS feature is not sufficient to explain the dramatic performance gains, as close to the same performance is exhibited with the feature disabled.)

- The STREAMS read notification feature allows the write side to exploit efficiencies from the knowledge of the amount of data that was requested by the read side. (However, this STREAMS feature is also not sufficient to explain the performance gains, as close to the same performance is exhibited with the feature disabled.)
- The STREAMS read fill mode feature permits the read side to block until the full read request is satisfied, regardless of the `O_NONBLOCK` flags setting associated with the read side of the pipe. (Again, this STREAMS feature is not sufficient to explain the performance gains, as close to the same performance is exhibited with the feature disabled.)
- The STREAMS flow control and scheduling mechanisms permits the read side to read more messages between wakeup events; and also permits the write side to write more messages between wakeup events. This results in superior code and data caching efficiencies and a greatly reduced number of context switches. This is the only difference that explains the full performance increase in STREAMS-based pipes over legacy pipes.

7 Conclusions

These experiments have shown that the *Linux Fast-STREAMS* implementation of STREAMS-based pipes outperforms the legacy Linux pipe implementation by a significant amount (up to a factor of 5) and outperform the *LiS* implementation by a staggering amount (up to a factor of 25).

> *The Linux Fast-STREAMS implementation of STREAMS-based pipes is superior by a significant factor across all systems and kernels tested.*

While it can be said that all of the preconceptions regarding STREAMS and STREAMS-based pipes are applicable to the under-performing *LiS*, and may very well be applicable to historical implementations of STREAMS, these preconceptions with regard to STREAMS and STREAMS-based pipes are dispelled for the high-performance *Linux Fast-STREAMS* by these test results.

- *STREAMS is fast.*

  Contrary to the preconception that STREAMS must be slower because it is more complex, in fact the reverse has been shown to be true for *Linux Fast-STREAMS* in these experiments. The STREAMS flow control and scheduling mechanisms serve to adapt well and increase both code and data cache as well as scheduler efficiency.

- *STREAMS is more flexible* and *more efficient.*

  Contrary to the preconception that STREAMS trades flexibility for efficiency (that is, that STREAMS is somehow less efficient because it is more flexible), in fact has shown to be untrue for *Linux Fast-STREAMS*, which is *both* more flexible *and* more efficient. Indeed, the performance gains achieved by STREAMS appear to derive from its more sophisticated queueing, scheduling and flow control model. (Note that this is in fitting with the statements made about 4.2BSD pipes being implemented with UNIX domain sockets for *"performance reasons"* [MBKQ97].)

- *Linux Fast-STREAMS is superior at exploiting parallelisms on SMP.*

  Contrary to the preconception that STREAMS must be slower due to complex locking and synchronization mechanisms, *Linux Fast-STREAMS* performed as well on SMP (hyperthreaded) machines as on UP machines and strongly outperformed legacy Linux pipes with 100% improvements at all write sizes and a staggering 500% at smaller write sizes.

- *STREAMS-based pipes are fast.*

  Contrary to the preconception that STREAMS-based pipes must be slower because STREAMS-based pipes provide such a rich set of features as well as providing full duplex operation where legacy pipes only unidirectional operation, the reverse has been shown in these experiments for *Linux Fast-STREAMS*. By utilizing STREAMS flow control and scheduling, STREAMS-based pipes indeed perform better than legacy pipes.

- *STREAMS-based pipes are neither unnecessarily complex nor cumbersome.*

  Contrary to the preconception that STREAMS-based pipes must be poorer due to their increased implementation complexity, the reverse has shown to be true in these experiments for *Linux Fast-STREAMS*. Also, the fact that legacy, STREAMS and 4.2BSD pipes conform to the same standard (POSIX), means that they are no more cumbersome from a programming perspective. Indeed a POSIX conforming application will not know the difference between the implementation (with the exception that superior performance will be experienced on STREAMS-based pipes).

- *LiS performs poorly.*

  Despite claiming to be an adequate implementation of SVR4 STREAMS, *LiS* performance is dismal enough to make it unusable. Due to conformance and implementation errors, *LiS* was already deprecated by *Linux Fast-STREAMS*, and these tests exemplify why a replacement for *LiS* was necessary and why support for *LiS* was abandoned by the OpenSS7 Project [SS7]. *LiS* pipe performance tested about half that of legacy Linux pipes and a full order of magnitude slower than *Linux Fast-STREAMS*.

8 Future Work

There are two future work items that immediately come to mind:

1. It is fairly straightforward to replace the pipe implementation of an application that uses shared libraries from underneath it using preloaded libraries. The *Linux Fast-STREAMS* `libstreams.so` library can be preloaded, replacing the pipe(2) library call with the STREAMS-based pipe equivalent. A suitable application that uses pipes extensively could be benchmarked both on legacy Linux pipes and STREAMS-based pipes to determine the efficiencies achieved over a less narrowly defined workload.
2. Because STREAMS-based pipes exhibit superior performance in these respects, it can be expected that STREAMS pseudo-terminals will also exhibit superior performance over the legacy Linux pseudo-terminal implementation. STREAMS pseudo-terminals utilize the STREAMS mechanisms for flow control and scheduling, whereas the Linux pseudo-terminal implementation uses the over-simplified approach taken by legacy pipes.

9 Related Work

A separate paper comparing a TPI STREAMS implementation of *UDP* with the Linux BSD Sockets implementation has also

been prepared. That paper also shows significant performance improvements for STREAMS attributable to the similar causes.

References

[GC94] Berny Goodheart and James Cox. *The magic garden explained: the internals of UNIX System V Release 4, an open systems design / Berny Goodheart & James Cox*. Prentice Hall, Australia, 1994. ISBN 0-13-098138-9.

[LfS] Linux Fast-STREAMS – A High-Performance SVR 4.2 MP STREAMS Implementation for Linux. http://www.openss7.org/download.html.

[LiS] Linux STREAMS (LiS). http://www.openss7.org/-download.html.

[LML] Linux Kernel Mailing List – Frequently Asked Questions. http://www.kernel.org/pub/linux/docs/-lkml/#s9-9.

[MBKQ97] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quaterman. *The design and implementation of the 4.4BSD operating system*. Addison-Wesley, third edition, November 1997. ISBN 0-201-54979-4.

[Rit84] Dennis M. Ritchie. A Stream Input-output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984. Part 2.

[SS7] The OpenSS7 Project. http://www.openss7.org/.

[Ste97] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison–Wesley, Reading, Massachusetts, fifteenth edition, December 1997. ISBN 0-201-56317-7.

A Performance Testing Script

A performance testing script (`perftest_sctipt`) was used to obtain repeatable results. The script was executed as:

```
$#> ./perftest_script -a -S10 --hiwat=$((1<<16)) --lowat=$((1<<13))
```

The script is as follows:

```
#!/bin/bash
set -x
interval=5
testtime=2
command=`echo $0 | sed -e 's,.*/,,'`
perftestn=
perftest=
if [ -x `pwd`/perftest ] ; then
  perftest=`pwd`/perftest
elif [ -x /usr/lib/streams/perftest ] ; then
  perftest=/usr/lib/streams/perftest
elif [ -x /usr/libexec/streams/perftest ] ; then
  perftest=/usr/libexec/streams/perftest
elif [ -x /usr/lib/LiS/perftest ] ; then
  perftest=/usr/lib/LiS/perftest
elif [ -x /usr/libexec/LiS/perftest ] ; then
  perftest=/usr/libexec/LiS/perftest
fi
if [ -x `pwd`/perftestn ] ; then
  perftestn=`pwd`/perftestn
elif [ -x /usr/lib/streams/perftestn ] ; then
  perftestn=/usr/lib/streams/perftestn
elif [ -x /usr/libexec/streams/perftestn ] ; then
  perftestn=/usr/libexec/streams/perftestn
elif [ -x /usr/lib/LiS/perftestn ] ; then
  perftestn=/usr/lib/LiS/perftestn
elif [ -x /usr/libexec/LiS/perftestn ] ; then
  perftestn=/usr/libexec/LiS/perftestn
fi
[ -n "$perftestn" ] || [ -n "$perftest" ] || exit 1
scls=
if [ -x `pwd`/scls ] ; then
        scls=`pwd`/scls
elif [ -x /usr/sbin/scls ] ; then
        scls=/usr/sbin/scls
fi
(
  set -x
  [ -n "$scls" ] && $scls -a -c -r pipe pipemod
  for size in 4096 2048 1024 512 256 128 64 32 16 8 4 2 1
  do
    [ -n "$perftest"  ] && $perftest  -q \
      -r -t $testtime -i $interval -m nullmod -p 0 -s $size ${1+$@}
    [ -n "$perftestn" ] && $perftestn -q \
      -r -t $testtime -i $interval -m nullmod -p 0 -s $size ${1+$@}
    [ -n "$scls"      ] && $scls -a -c -r pipe pipemod srvmod nullmod
  done
) 2>&1 | tee `hostname`.$command.`date -uIseconds`.log
```

B Raw Data

Following are the raw data points captured using the *perftest_script* benchmarking script:

Table 1 lists the raw data from the `perftest` program that was used in preparing graphs for FC6 (i386) on Porky.

Table 2 lists the raw data from the `perftest` program that was used in preparing graphs for CentOS 4 on Porky.

Table 3 lists the raw data from the `perftest` program that was used in preparing graphs for SuSE OSS 10 on Porky.

Table 4 lists the raw data from the `perftest` program that was used in preparing graphs for Ubuntu 6.10 on Porky.

Table 5 lists the raw data from the `perftest` program that was used in preparing graphs for RedHat 7.2 on Pumbah.

Table 6 lists the raw data from `perftest`, used in preparing graphs for Fedora Core 6 (x86_64) HT on Daisy.

Table 7 lists the raw data from `perftest`, used in preparing graphs for CentOS 5 (x86_64) HT on Daisy.

Table 8 lists the raw data from `perftest`, used in preparing graphs for CentOS 5.2 (x86_64) HT on Daisy.

Table 9 lists the raw data from `perftest`, used in preparing graphs for SuSE 10.0 OSS on Mspiggy.

Size	LiS	STREAMS	Linux
1	37188	344307	116966
2	37284	351804	117820
4	37179	347164	116381
8	37030	338055	117887
16	37225	329919	117822
32	36999	317133	116595
64	36809	302554	116686
128	35127	283041	117284
256	34828	271630	114657
512	34807	263021	114821
1024	34607	247080	111825
2048	34204	214279	106369
4096	33139	176842	100510

Table 1: Raw data for Fedora Core 6 on Porky

Size	LiS	STREAMS	Linux
1	53119	479434	132195
2	53066	505597	132293
4	53289	501230	131201
8	53216	475951	132182
16	53254	464013	131688
32	52952	438519	131697
64	52499	407751	129409
128	50065	379356	130188
256	49348	372393	126861
512	49297	360773	125318
1024	48598	336727	123318
2048	48274	290614	117809
4096	47004	227778	110875

Table 2: Raw data for CentOS 4.0 on Porky

Size	LiS	STREAMS	Linux
1	53119	961820	168049
2	53066	933673	176267
4	53289	942865	172912
8	53216	837034	168898
16	53254	827399	166427
32	52952	740263	172185
64	52499	659878	169231
128	50065	582512	174005
256	49348	580011	166646
512	49297	547149	167829
1024	48598	512452	152447
2048	48274	413858	154813
4096	47004	307174	138756

Table 3: Raw data for SuSE 10.0 OSS on Porky

Size	LiS	STREAMS	Linux
1	53119	430184	144855
2	53066	433274	143835
4	53289	425094	145879
8	53216	407647	143399
16	53254	394244	141268
32	52952	372063	144056
64	52499	354598	139854
128	50065	339602	141793
256	49348	324405	140269
512	49297	311610	134445
1024	48598	292892	136385
2048	48274	255374	127651
4096	47004	202755	116218

Table 4: Raw data for Ubuntu 6.10 on Porky

Size	LiS	STREAMS	Linux
1	53160	497439	209223
2	53440	499519	199566
4	53252	496272	187440
8	53097	489615	188829
16	53179	485036	182148
32	52926	469102	185174
64	53535	457550	182383
128	49452	416632	178087
256	49584	396356	177204
512	49169	381209	165517
1024	48992	355111	173222
2048	47970	303334	163572
4096	46598	240386	136522

Table 5: Raw data for RedHat 7.2 on Pumbah

Size	LiS	STREAMS	Linux
1	37188	334896	146553
2	37284	334796	122048
4	37179	329476	140025
8	37030	341612	160396
16	37225	333520	125678
32	36999	325169	125124
64	36809	302603	109340
128	35127	278490	128133
256	34828	247379	122689
512	34807	235190	104739
1024	34607	215718	83447
2048	34204	187982	81301
4096	33139	150392	77118

Table 6: Raw data for Fedora Core 6 on Daisy

Size	STREAMS	Linux
1	232029	262762
2	221413	224493
4	217800	294765
8	206325	284109
16	194132	252365
32	200675	347247
64	198463	290958
128	210551	280954
256	168482	270703
512	164051	198930
1024	154924	197374
2048	138744	101038
4096	116291	70827

Table 7: Raw data for CentOS 5 on Daisy

Size	STREAMS	Linux
1	693675	133714
2	677583	192987
4	652141	137415
8	617148	153168
16	544888	127430
32	430115	193361
64	322424	118036
128	221081	143728
256	187469	110754
512	185642	102351
1024	174350	103238
2048	155567	106183
4096	130076	77321
8192	68071	62987

Table 8: Raw data for CentOS 5.2 on Daisy

Size	LiS	STREAMS	Linux
1	690896	114253	
2	662870	107000	
4	639931	128916	
8	596668	114321	
16	520846	124913	
32	427005	132284	
64	321028	105085	
128	212760	103989	
256	177236	71573	
512	173048	58217	
1024	163703	66249	
2048	154194	64369	
4096	134026	63115	
8192	66860	55217	

Table 9: Raw data for SuSE 10.0 OSS on Mspiggy