The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

# An Analysis of Process and Memory Models to Support High-Speed Networking in a UNIX Environment

B. Murphy, University of Cambridge
S. Zeadally and C. J. Adams, University of Buckingham

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

# An Analysis of Process and Memory Models to Support High-Speed Networking in a UNIX Environment

*BJ Murphy*
*Computer Laboratory, University of Cambridge, UK*
*S Zeadally, CJ Adams*
*Department of Computer Science, University of Buckingham, UK*

**Abstract**

In order to reap the benefits of high-speed networks, the performance of the host operating system must at least match that of the underlying network. A barrier to achieving high throughput is the cost of copying data within current host architectures. We present a performance comparison of three styles of network device driver designed for a conventional monolithic UNIX kernel. Each driver performs a different number of copies. The zero-copy driver works by allowing the memory on the network adapter to be mapped directly into user address space. This maximises performance at the cost of: 1) breaking the semantics of existing network APIs such as BSD sockets and SVR4 TLI; 2) pushing responsibility for network buffer management up from the kernel into the application layer. The single-copy driver works by copying data directly between user space and adapter memory obviating the need for an intermediate copy into kernel buffers in main memory. This approach can be made transparent to existing application code but, like the zero-copy case, relies on an adapter with a generous quantity of on-board memory for buffering network data. The two-copy driver is a conventional STREAMS driver. The two-copy approach sacrifices performance for generality. We observe that the STREAMS overhead for small packets is significant. We report on the benefit of the hardware cache in ameliorating the effect of the second copy, although we note that streaming network data through the cache reduces the level of cache residency seen by the rest of the system.

A barrier to achieving low jitter is the non-deterministic nature of many operating system schedulers. We describe the implementation and report on the performance of a kernel streaming driver that allows data to be copied between a network adapter and another I/O device without involving the process scheduler. This provides performance benefits in terms of increased throughput, increased CPU availability and reduced jitter.

## 1. Introduction

The integrated support of distributed continuous media systems poses design issues for the architecture of the network over which such a system is distributed and for the operating system running in the end-user terminals. This paper addresses the subject of operating system support for audio and video, particularly focussing on the reduction of delay and jitter.

The UNIX operating system was not designed to provide optimal support for continuous media. Traditionally, fair sharing of resources between multiple users has been the prime concern. However, as UNIX moves from main-frame to desktop, high throughput and deterministic response times become increasingly important.

This paper considers a number of techniques that are relevant to the support of high-speed networks in general and continuous media in particular within a conventional, and unmodified, monolithic UNIX kernel. We make two contributions. Firstly, we present the design of a network interface based on a zero-copy memory model, we compare its performance with single-copy and conventional two-copy models, and we discuss compatibility, protection and other implications. Secondly, we present the design of a process model in which data is streamed between device drivers without traversing the user-kernel boundary. Our motivation was to explore the options for integrating continuous media into a traditional operating system.

## 2. Architectural Background

### 2.1. Data Copying

The most important factor that limits application-network throughput in current generation workstations is the high cost of copying data relative to the cost of processing that data. This is because improvements in memory performance have not kept pace with improvements in processor performance [11]. Furthermore, improvements in network technology now mean that memory bandwidth is often within the same order of magnitude as network bandwidth. In order to exploit the throughput potential of the network, therefore, the number of copies between application and network adapter must be kept to a minimum.

A conventional network subsystem in a monolithic kernel such as UNIX requires two copies of the data in both the transmit and receive directions: one copy is required between application and kernel; an additional copy is required between kernel and network adapter. The network subsystem generally performs some level of protocol processing depending on the requirements of the application. For example, in order to provide a reliable byte-stream service, the network subsystem implements a communications protocol such as TCP which performs flow control and error recovery. The network subsystem manages kernel buffers according to the specification of the protocol in order to handle functions such as packet retransmission and re-assembly.

One opportunity to improve performance is to eliminate the copy from kernel to network adapter. This requires that the network adapter be equipped with on-board memory which can be mapped into host address space. In the transmit direction, the network subsystem copies data from application address space directly into adapter memory. Protocol processing may be performed and protocol headers inserted before the adapter is instructed to transmit the packet. In the receive direction, the network adapter assembles an incoming packet in adapter memory. The network subsystem may then perform protocol processing on the packet before copying the data directly into application address space. This is the approach taken by the designers of the Medusa FDDI adapter [1] and the Afterburner [3] TCP/IP accelerator board. A further optimisation is the addition of hardware assist for on-the-fly checksum calculations as the data is copied between kernel and user space. The only changes that are required are to the network subsystem; existing applications reap the benefits of reduced copying without change.

Another opportunity to improve performance is to eliminate the copy between application and kernel. A number of virtual memory (VM) techniques have been proposed including page-remapping (in which pages are unmapped from one protection domain and mapped into the other) and copy-on-write (in which pages are shared between protection domains and copying is delayed until a process in one domain attempts to write to a shared page). These techniques are particularly relevant to micro-kernel architectures in which data may traverse multiple protection domains [5]. An alternative technique for eliminating the user-kernel copy is by the use of statically shared memory. All of these techniques normally require some degree of co-operation from the application (for example, using page-aligned buffers or pinning pages into memory) as well as possible changes to the VM and network subsystems.

By themselves, each approach (eliminating kernel-adapter copies and eliminating user-kernel copies) provides single-copy transfers between application and network. In combination, and with the appropriate hardware, zero-copy transfers are achievable.

### 2.2. Context Switches

Another factor that limits application-network throughput is the cost of a context switch. Normally, a hardware interrupt is generated every time a packet arrives from the network[1]. On a heavily loaded system, each such interrupt will be accompanied by a context switch from the currently running user process to the user process for which the packet is intended. Context switches between user processes are expensive. The explicit overhead involves the saving of one context (register contents, memory management information, etc) and the restoration of another. However, there is an implicit cache-performance cost that, depending on the host architecture, may dominate other overheads [14].

Apart from these throughput overheads, context switches can have a serious impact on the

---

**1** An intelligent network adapter may re-assemble lower-level protocol data units (such as ATM cells) in order to reduce the interrupt load on the host.

*timely* delivery of data. Traditional UNIX systems provide no bounds on context switch latency. This is because a process running in kernel mode cannot be pre-empted. Furthermore, the conventional time-sharing scheduling policy leads to non-deterministic response times. Modern versions of UNIX (such as System V Release 4) attempt to alleviate these problems by providing a pre-emptible kernel together with real-time scheduling classes [10]. Unfortunately, early experiences of such systems have not been encouraging [15]. Applications that involve the transfer of jitter-sensitive continuous media between devices may be better served by a mechanism that completely eliminates the need for context switches.

Context switches can be avoided by exploiting kernel-level or hardware-level streaming [4]. Kernel-level streaming involves the transfer of data from source to sink over the system bus without an application process being included in the data path. Since the data passes through memory, data manipulations by the CPU are possible. Hardware-level streaming involves the transfer of data from source to sink over a private data bus or by peer-to-peer DMA over the system bus. No CPU manipulations of the data are possible. These two types of streaming contrast with the more conventional user-level streaming arrangement in which a user process is involved in the data path.

Our interest in kernel-level streaming stems from the observation that many continuous media applications involve the transfer of data between a network adapter and another device without requiring the *manipulation* of that data. For example, a server process may move video between network and disk; a client process may move audio between network and codec. In these cases, there is no requirement to touch the data beyond the presentation-layer conversions performed by the protocol stack. The fact that the data needs to cross the user-kernel boundary (twice) is an artefact of the design of the I/O sub-system. Performing the transfer in the kernel reduces the amount of copying involved and eliminates the context-switch overhead that would otherwise be incurred. This improves throughput, improves CPU availability, and reduces jitter.

# 3. Implementation

## 3.1. Copy Avoidance

Three different styles of device driver (two-copy, one-copy and zero-copy) have been implemented for the CHARISMA Asynchronous Transfer Mode (ATM) network adapter. ATM is a switching and multiplexing scheme based on the use of small fixed-size "cells" which is likely to be deployed in many high-speed networks of the future [17]. The CHARISMA ATM adapter has been designed for the EISA bus and features 1 MB of dual-ported RAM which is memory-mapped into host address space. The on-board 25 MHz T801 transputer offers an ATM Adaptation Layer 5 (AAL5) interface to the host processor. AAL5 provides a variable-length packet transfer service with error detection on top of the fixed-length cell relay service provided by ATM [17]. All communication between the two processors is performed by the manipulation of a pair of software FIFOs in shared memory and by hardware interrupts. As noted in [3, 6, 13], the use of a pair of shared memory FIFOs to pass buffer descriptors coupled with the atomicity of load and store instructions allow two processors to exchange data without the need for synchronisation primitives. Except for the small amount of space used by these FIFOs and ancillary control structures, the shared memory is organised as two pools of fixed-sized buffers (4 KB, equal to the VM page size), one pool for transmission and the other for reception. Data is transferred using memory load/store instructions. The CHARISMA host adapter is described in detail in [13].

The two-copy "`atm`" driver is a STREAMS [19] driver that provides both a DLPI [24] interface and a "raw" interface to the underlying AAL5 service. Within the STREAMS model, device drivers, modules and multiplexors interact by passing messages between uni-directional queues. A STREAM head provides user processes with the point of access to a particular stream (queue-pair). No copying is involved as data is passed between queues since messages are passed by reference. The DLPI interface allows the driver to be linked under the standard IP multiplexor thereby supporting the transmission of IP packets over our ATM network. The raw interface allows a user process to transmit and receive AAL5 packets using the `read()` and `write()` system calls. The `atm` driver provides two-copy transfers between application and network.

The single-copy "cpatm" driver is a non-STREAMS character driver that simply copies data between application and network. It provides a read()/write() interface to the underlying AAL5 service. The cpatm driver provides single-copy transfers between application and network but does not permit integration with in-kernel network protocols[2].

The zero-copy "mmatm" driver is a non-STREAMS driver that eliminates copies between a co-operating application and the network. It supplies a mmap() entry point that allows an application process to map buffers (pages) from adapter memory directly into user address space. This allows the data path between application and network to bypass the kernel entirely. The control path is managed by the mmatm driver. Before an application can transmit a packet, it must make an ioctl() request to the driver to obtain a buffer identifier. The buffer identifier specifies the area of adapter memory that has been allocated to the application. The application then writes its data directly into the specified buffer on the adapter. In order to transmit the data, the application makes an ioctl() call to the driver passing the buffer identifier as the argument. In order to receive a packet, the application makes a (potentially blocking) ioctl() call; when a packet arrives, the driver passes back the buffer identifier as a return parameter to this call. The application consumes the data and makes an ioctl() call to release the buffer. Notice that the mmatm driver does not provide read() or write() entry points. Notice also that two system calls (one for buffer management and one for data transfer) are required to transfer a packet in each direction. These issues are discussed later.

## 3.2. Context-Switch Avoidance

The kernel streaming "kproc" driver is a STREAMS multiplexor under which two device drivers may be linked. The kproc driver can be instructed to copy data from the read queue of one driver to the write queue of the other, and vice-versa (Figure 1). This is exactly the behaviour of a STREAMS-based IP multiplexor that has been configured to act as a router. The presence of the

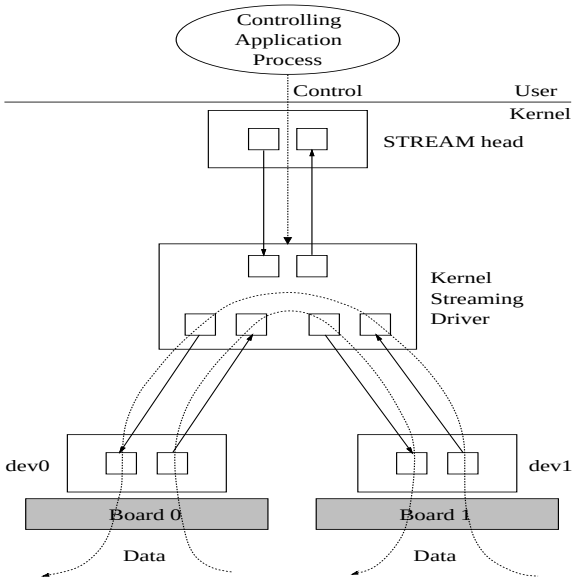


Figure 1: STREAMS-based Kernel-level Streaming Implementation

kproc driver is transparent to the devices linked below it; each driver is unaware of whether its queues are connected to a user process or to another driver. The kproc driver allows data to be transferred between a pair of devices using only two copies instead of four (since the two copies across the user-kernel boundary are eliminated) and without any context switches.

A key feature of this model is the separation of the control path from the data path [18]. The data path is within the kernel, while an application process may control the flow by means of an out-of-band channel. This is a radical departure from the conventional UNIX I/O model. Although our kproc driver does nothing more than transfer data between devices, it would be possible to perform in-band processing should that be required. For example, flow-specific events (such as video frame boundaries) could be signalled to the application in order to support higher level synchronisation services. Furthermore, additional STREAMS modules could be "pushed" onto the data path to perform higher level protocol processing such as transport-level control and presentation-level format conversions.

---

**2** A STREAMS procedure (esballoc()) enables a device driver to eliminate the copy from a memory-mapped adapter to a kernel buffer by allowing a STREAMS message to be wrapped around a user-supplied buffer. (One of the arguments to esballoc() is a pointer to a function to be called to de-allocate the buffer.) We suggest that the SVR4 STREAMS implementation be modified to provide the STREAM head with the same facility. This would permit the implementation of an integrated single-copy STREAMS-based network subsystem in both the receive *and* transmit directions. None of our experiments described here made use of the esballoc() procedure.

## 4. Experimental Detail

In order to evaluate the performance of these models, a simple traffic generator/receiver program was written for the T801 transputer on the network adapter. The program can be instructed (via a shared memory interface) to generate a series of packets of specified size and at a specified rate or to sink a series of packets. In combination with an equivalent program running as a UNIX user process, measurements were taken to determine the performance of each driver. Each program has access to the 1 μs clock provided by the T801; the transputer program copies the value of the clock into shared memory (at a granularity of approximately 4 μs) for the benefit of the UNIX program. The programs timestamp outgoing and incoming packets in local memory. At the end of a test run, the transputer program is instructed to dump its array of timestamps into shared memory. Elapsed times are calculated by simply subtracting corresponding timestamps. These values represent the time taken to transfer a series of packets between a network adapter and a UNIX user-level process.

In order to measure the performance of the kernel streaming driver, we equipped the host with two of our network adapters. Each board runs a separate copy of the transputer test program. A UNIX control program first links the two instances of the device driver below the `kproc` multiplexor. It then configures the transputer program on one board to act as a sink of packets and the other transputer program to act as a source of packets. At the end of the test sequence, the control program instructs each transputer to dump its array of timestamps. Elapsed times are calculated allowing for the offset between the two clocks. These values represent the time taken to transfer a series of packets between a network adapter and another hardware device using kernel-level streaming.

Two variations of kernel-level streaming were investigated: synchronous streaming, in which packets are transferred directly by the interrupt handler, and asynchronous streaming, in which packets are only queued by the interrupt handler and transferred some time later by the STREAMS scheduler (with most interrupts enabled). It is possible to configure the style of message passing (synchronous or asynchronous) on a per-stream basis by means of an `ioctl()` call.

The kernel streaming configuration just described was compared with a conventional user-level streaming configuration. A UNIX user-level streaming program was used to read packets from one board and write packets to the other board in the traditional manner using the "raw" `read()/write()` interface provided by the STREAMS `atm` driver. The streaming program was run as both a time-sharing and a real-time process.

In order to gain a proper understanding of the relative merits of kernel-level streaming versus user-level streaming, we conducted these experiments under a number of different operating conditions. The first set of readings was taken without artificially loading the system. A second set of readings was taken while running a compute-intensive "soak" program comprising a tight loop. The "soak" program ensures that the processor is never idle. It spends no time in the kernel. A third set of readings was taken while running an I/O-intensive "find" program comprising a recursive search of an NFS filing system over Ethernet. The "find" program spends a considerable proportion (over 95%) of its execution time in the kernel performing system calls. Furthermore, the "find" program causes interrupts to be generated by the Ethernet adapter. It spends approximately 85% of the time sleeping (waiting for I/O). The interrupt priority level of the Ethernet driver was the same as that of our ATM driver. The "soak" and "find" processes were run in the default time-sharing scheduling class.

All our measurements were made on a 33 MHz Intel 80486 EISA machine running an otherwise unloaded UNIX System V Release 4.2 kernel in the default multi-user state. The machine features 32 MB of RAM, 8 KB of physically-indexed four-way set associative write-through primary cache, and 256 KB of physically-indexed two-way set associative write-back secondary cache.

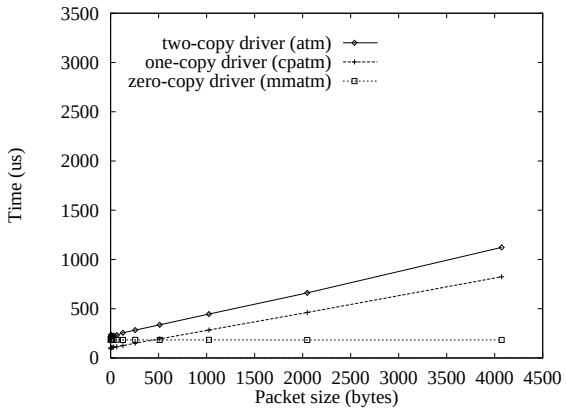


Figure 2: Driver Transmit Performance (Caching Enabled)

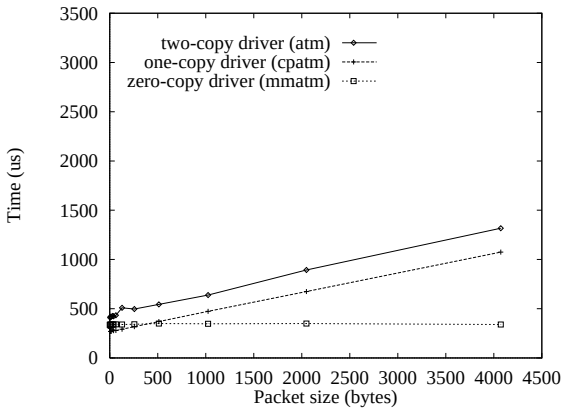


Figure 3: Driver Receive Performance (Caching Enabled)

# 5. Results

## 5.1. Comparison of Device Driver Performance

Figures 2 and 3 illustrate the performance of the three types of device driver for different sizes of packet. The data points represent the median of 1000 timing measurements for each size of packet using an inter-packet transmission interval of 10 ms.

### *5.1.1. Relative Cost of Transmission versus Reception*

The first point to notice is that the cost of reception is approximately twice that of transmission for all three types of driver. This is typical of network subsystems in general. It is explained by the fact that whereas packets are transmitted synchronously with respect to the source they are received asynchronously with respect to the sink. This normally demands that the operating system handle an interrupt, wakeup the waiting process and possibly switch context for every incoming packet.

### *5.1.2. Relative Performance of Zero-Copy versus Single-Copy Drivers*

The second point to observe is that for small packets (less than approximately 500 bytes) the performance of the single-copy `cpatm` driver is better than that of the zero-copy `mmatm` driver. This is because we have included the cost of the system call required to perform buffer management for each packet transfer in our readings for the zero-copy driver. Recall that an `mmatm` user wishing to transmit a packet must first ask the driver for a free buffer. Likewise, when an `mmatm` user has finished processing an incoming packet, it must return the buffer to the driver. This additional overhead is included in our analysis in order to permit a fair comparison.

### *5.1.3. Relative Cost of STREAMS versus non-STREAMS Drivers*

As expected, the STREAMS `atm` driver provides the worst performance. There are two explanations for this. Firstly, the STREAMS driver performs two copies of the data. Secondly, there is an inherent overhead associated with STREAMS as indicated by the difference between the points of intersection on the ordinate for the STREAMS `atm` and the non-STREAMS `cpatm` drivers. That is, for very small packets for which the overhead associated with copying is negligible, the performance of the STREAMS driver is worse (by a factor of approximately 2) than that of the non-STREAMS driver. Part of the reason for this is that every packet on a stream takes the form of a STREAMS message with an associated header that requires allocation and de-allocation by the driver and STREAM head. A further reason is that the movement of a message between driver and application involves processing by the STREAM head and possibly by the STREAMS scheduler in addition to the driver. The inherent STREAMS overhead becomes less significant with increasing packet size as the copying overhead becomes dominant.

### *5.1.4. Effects of the Cache*

The final point to notice is that the performance of the two-copy driver does not degrade with packet size as quickly as one might anticipate

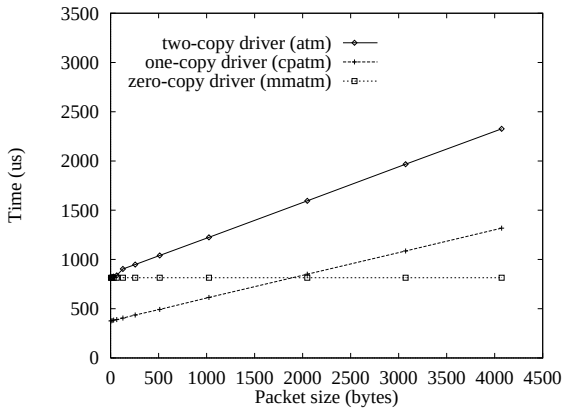


Figure 4: Driver Transmit Performance
(Caching Disabled)

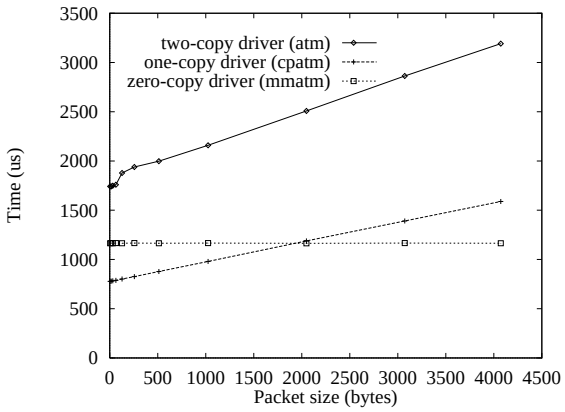


Figure 5: Driver Receive Performance
(Caching Disabled)

given the extra copy involved relative to the single-copy driver. This is illustrated by the fact that the gradient of the lines relating to the two-copy `atm` driver is very nearly the same as the gradient of the lines relating to the single-copy `cpatm` driver. We speculated that this was due to the hardware cache, but decided to investigate further. Figures 4 and 5 show a repeat of the experiments with all caches (primary and secondary) disabled. Under these conditions, the throughput characteristics of the two-copy `atm` driver demonstrate the cost of the extra (un-cached) copy. Specifically, the gradient of lines relating to the two-copy `atm` driver is greater (by a factor of approximately 1.7) than the gradient of lines relating to the single-copy `cpatm` driver. With caches on (Figures 2 and 3), the effect of the copy between main memory and main memory (across the user-kernel boundary) is ameliorated by the fact that the data is already cached. This degree of cache residency is unlikely to be achievable under realistic load conditions in which manipulation of network buffers is interleaved with memory accesses from other processes. Furthermore, the benefit of the cache to a network subsystem that performs multiple copies is at the expense of reduced cache residency seen by other processes in the system. Neither of these drawbacks are revealed by our experiments.

The cost of the copy between adapter memory and main memory is almost independent of whether or not caches are enabled. This is illustrated by the fact that the gradient of the lines relating to the single-copy `cpatm` driver in Figures 2, 3, 4 & 5 is virtually constant. This is because the adapter memory is configured to be non-cacheable to prevent cache consistency problems [20]. The cost of accessing adapter memory over the system bus dominates the cost of accessing main memory regardless of whether or not caches are enabled.

## 5.2. Comparison of User-level Streaming versus Kernel-level Streaming

Figures 6, 7 and 8 illustrate the performance of the various streaming implementations under three operating conditions. We have adopted the graphical presentation format devised by Faller [9]. The ordinate shows the time taken to transfer a 4072-byte block (the maximum size of an AAL5 service data unit that fits into a 4 KB buffer) between one network adapter and the other. The abscissa shows the percentage cumulative frequency of these transfer times. The value on the ordinate corresponding to *P*% on the abscissa is called the *P*th percentile of the distribution. For example, an 80th percentile of 2 ms means that 80% of the measured transfer times were at or below 2 ms. The value of the 50th percentile is, by definition, the median. A flat graph (ie a horizontal line) represents zero variance. We have used a logarithmic scale on the ordinate in order to dampen the visual distortion that would otherwise be caused by a small number of large readings. We have used an exponential scale on the abscissa in order to highlight the relatively small number of readings of particular interest. Each test run comprised the transfer of 1000 packets with an inter-packet transmission time of 10 ms. The results are plotted at 1% intervals on the abscissa. The slight positive gradient that is observable in the graphs (approximately 130 µs over the 10 s measurement period) is due to drift between the two transputer clocks.

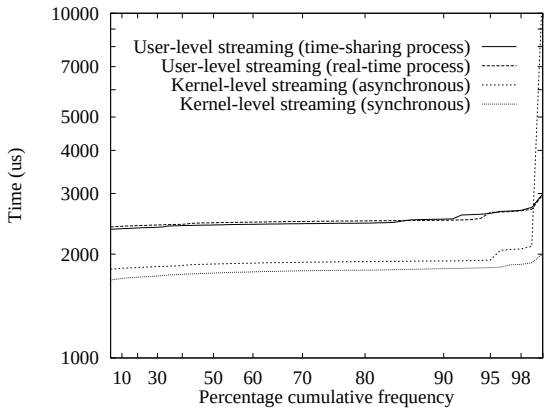


Figure 6: Streaming Performance with System Idle

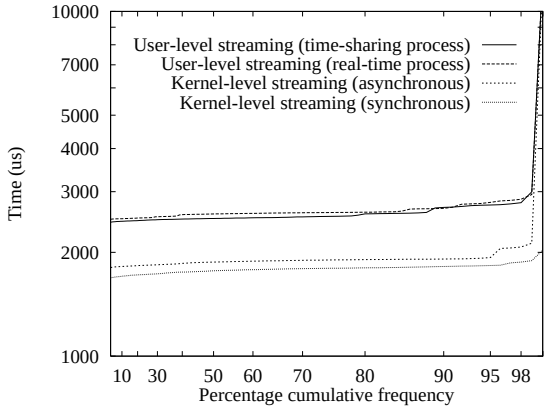


Figure 7: Streaming Performance with Compute-Intensive Load

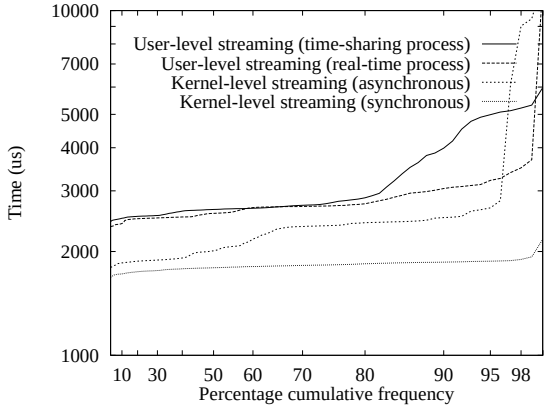


Figure 8: Streaming Performance with I/O-Intensive Load

*5.2.1. Kernel-level Streaming with no Artificial Load*

Figure 6 shows the performance of the four streaming implementations with no artificial load. Kernel-level streaming provides a throughput increase of approximately 25%. Furthermore, average CPU utilisation (as measured by the public domain `top` program) is reduced by between 30% and 50% depending on whether the transfer is performed by the STREAMS scheduler or by the interrupt handler respectively. A comparison of the graphs for the two variations of kernel-level streaming demonstrates the overhead imposed by the STREAMS scheduler (approximately 5%).

*5.2.2. Kernel-level Streaming with Compute-Intensive Load*

Figure 7 shows the minimal impact of the compute-intensive load on the two user-level streaming models. The only performance penalty is the cost of pre-empting the "soak" process for each packet transfer. This overhead (which is just discernible from a comparison of Figures 6 and 7) is approximately 90 µs. The reason why the "soak" process does not have a more serious impact on the performance of the time-sharing user-level streaming process is that, although both are competing for the processor within the same scheduling class, the scheduler ensures that the priority of the I/O-bound process remains higher. This policy is to help provide better response times for interactive processes compared with compute-bound processes. There is no discernible reduction in the performance of the kernel-level streaming implementations.

*5.2.3. Kernel-level Streaming with I/O-Intensive Load*

Figure 8 shows the impact of an I/O-intensive load. The real-time user-level streaming process displays much better jitter characteristics than the time-sharing process. This is because pre-emption points within the kernel allow the real-time streaming process to pre-empt the "find" process while the latter is in the kernel. The synchronous implementation of kernel-level streaming is virtually unaffected by the extra I/O load; this is significant, but not surprising since all the work is being done in the interrupt handler. The asynchronous implementation of kernel-level streaming performs badly for approximately 5% of the data set; we are unable to explain this behaviour.

The maximum throughput of the kernel-level streaming model (approximately 18 Mbits/s), which

involves two copies of the data, is bounded by adapter memory to main memory bandwidth (approximately 40 Mbits/s). It is reasonable to speculate that higher rates would be achievable using faster memory.

# 6. Discussion

## 6.1. Copy Avoidance

The elimination of all copies (using the shared memory technique described here), whilst attractive for performance reasons, raises a number of issues that we address below.

### *6.1.1. Hardware Constraints*

Not all network adapters offer a memory-mapped interface without which copy elimination is impossible. Many peripherals reside on an I/O bus and make use of DMA or programmed I/O for data transfer; the choice between the two is often dictated by the host bus architecture [1]. We suggest that the potential performance benefits of zero-copy transfers need to be taken into account when making the choice between the memory bus and the I/O bus during the adapter design stage and when evaluating the merits of competing host bus architectures.

An additional problem is that on-board memory is a limited resource. Buffering protocol data (in order to provide single-copy transfers) or even application data (in order to provide zero-copy transfers) in adapter memory rather than main memory may not be realistic for high volume flows. Nevertheless, where memory-mapped designs are applicable, while the discrepancy between CPU performance and memory performance remains, and while the cost of memory continues to fall, we believe that there is a strong case for furnishing the network adapter with generous quantities of RAM.

### *6.1.2. Application Program Interface*

Current network APIs, such as BSD sockets and SVR4 TLI, allow applications to allocate arbitrarily aligned buffers located anywhere in process address space and which are contiguous in virtual (but not necessarily physical) address space. The cost of this amenity is generally an extra copy by the network subsystem thereby limiting the scope for copy avoidance. We endorse a model in which network buffers are allocated by, and at the convenience of, the network subsystem rather than the application [4]. Responsibility for buffer de-allocation is shared between the network subsystem and the application. The transmission paradigm becomes one in which the application requests a buffer from the network subsystem, writes to the buffer, and then requests transmission. The reception paradigm becomes one in which the application requests reception of data (optionally specifying a limit on size), is handed a buffer by the network subsystem, consumes the data and de-allocates the buffer. A network buffer should be represented as an abstract data type rather than a single contiguous buffer [12]. This leaves the network subsystem free to implement such types in an optimal manner.

The advantage of such an API is its generality. Where hardware permits, it allows zero-copy transfers. Where hardware or other constraints rule out copy elimination, it facilitates single-copy transfers without having to "bend" existing APIs (by mandating page-aligned buffers, for example). We suggest that user-level protocols [7, 23] would benefit from a new API: the semantics of existing APIs are likely to force a copy even when the protocol is implemented as a user-level library and no protection domains are being crossed.

The disadvantage of such an API is that it is incompatible with current network APIs and therefore cannot be of benefit to the great many existing applications that are based on these APIs. Nevertheless, we believe that there is a new class of applications that could benefit from the performance improvements made possible by an API that is better integrated with the network subsystem. Clearly, a variety of APIs can exist side by side: continuous media applications would benefit from the new API and backward compatibility could be provided by traditional APIs at the cost of reduced performance.

### *6.1.3. Protection*

A ramification of this new API is that responsibility for the de-allocation of buffers used by the network subsystem is pushed up from the kernel into the application layer. Furthermore, there are protection implications when such an API is underpinned by a shared memory implementation such as that described here. In order to circumvent these problems, buffers need to be managed as pools each of which is owned by a separate process. The size of each pool could be based on the throughput requirements of the application as specified at call setup time or on the dynamics of the flow subject to a system-specified per-application limit. The device driver would permit the application to map in only the

shared memory pages belonging to its pool. Thereafter, the VM subsystem ensures that an application cannot interfere with data belonging to another process. A malicious or badly programmed application that does not free its buffers only exhausts its private pool and does not compromise the operation of the entire system.

#### *6.1.4. Operating System Integration*

The zero-copy driver discussed here does not add value to the network service provided by the network adapter. Higher-level protocols could be implemented in user-space based on this design. If it is necessary to implement such protocols in the kernel, however, changes need to be made to the operating system to enable the network subsystem to handle the abstract buffer type outlined above.

#### *6.1.5. Early Demultiplexing*

A key requirement in this design is that an early demultiplexing decision can be made on incoming data [4]. In ATM networks, the VCI is ideal for this purpose provided that multiple higher-level protocols are not multiplexed over the same channel [22].

#### *6.1.6. Performance For Small Packets*

A problem with our zero-copy implementation is that, for small packets, the buffer management overhead outweighs the benefits of copy elimination. This is due to the cost of making a system call. A solution is to employ a shared memory interface between application and driver for the purpose of buffer management much like the shared memory interface between driver and adapter for the purpose of data transfer. Other researchers have noted this problem and propose a similar solution [21].

#### *6.1.7. Cache Performance*

A feature of the zero-copy model presented here is that network data is not brought into the cache unless and until it is explicitly copied by the processor. This hurts protocol stack implementations that perform multiple touches of the data since all accesses to network buffers go over the bus. However, there are a number of benefits. Firstly, the level of cache residency seen by the rest of the system increases if network data does not enter the cache. Secondly, incoming network data is only brought into the cache if and when the application consumes the data (ie as late as possible). This maximises cache residency by eliminating the potential for context switches between the data being brought into the cache (by the network subsystem) and the data being consumed by the application [16]. Note that the performance penalty incurred by making non-cacheable accesses to adapter memory is reduced with protocols that touch only part of a packet (eg the header) rather than the entire packet. Such protocols generally sacrifice error detection (by eliminating the checksum, for example), but many "raw" multimedia flows are resilient to some degree of data corruption. Furthermore, implementation strategies based on integrated layer processing [2] ensure that the cost of accessing adapter memory is kept to a minimum.

### 6.2. Context-Switch Avoidance

Kernel-level streaming has the potential for delivering increased throughput, increased CPU availability, and reduced jitter. In order to realise these benefits in a generic fashion, however, the entire I/O sub-system needs restructuring. For example, in order to be able to stream data directly from disk to the network, the filing system needs to offer an appropriate interface. Nevertheless, we believe that this technique is applicable to a large class of multimedia I/O devices.

An added feature of this model is that careful implementations can avoid streaming continuous media through the cache. For example, if an adapter is equipped with on-board memory, data could be transferred from that device by copying pointers to the data (in the form of STREAMS messages wrapped around adapter buffers, for example) rather than the data itself. Alternatively, data could be transferred between adapters using DMA via main memory and, provided that protocol processing does not require touching the entire packet, the impact on the cache is minimised.

The concepts of kernel-level and hardware-level streaming are not new [4, 8, 18]. We believe that kernel-level streaming is a good compromise between between hardware-level streaming and user-level streaming. Hardware-level streaming does not involve the processor: all manipulation of the data, including protocol processing, must be performed by intelligent peripherals. User-level streaming is the most flexible but incurs the highest costs in terms of performance and resource usage. We observe that, unlike other implementations [8], the STREAMS I/O sub-system

supports kernel-level streaming without requiring modifications to the operating system.

We have not addressed the issue of streaming *multiple* flows between devices within the kernel. Under such circumstances, there will be "cross-talk" between the flows. The FIFO scheduling discipline provided by our prototype implementation does not serve the needs of multiple flows with conflicting "quality of service" requirements. In passing, we note that the STREAMS scheduler supports multiple priority levels which could be exploited in a more sophisticated implementation to minimise cross-talk. Nevertheless, we believe that a better solution would be an operating system that provided a single integrated scheduling mechanism for all CPU activity including that related to inter-device flows, conventional application-terminated flows, and compute-bound tasks.

## 7. Conclusions

We have reported on the performance characteristics of a number of process and memory models to support high-speed networks within a monolithic UNIX kernel. We have presented the design of a zero-copy device driver for an ATM network adapter. The design uses shared memory which is mapped directly into user address space. We have demonstrated the performance benefits and discussed the implications of such a design.

We suggest that existing APIs, such as BSD sockets and SVR4 TLI, are a barrier to achieving high network throughput. We advocate a new style of API in which buffers are allocated by the network subsystem rather than by the application. We suggest that such an API improves the scope for copy avoidance.

We have demonstrated that streaming data between two devices in the kernel provides performance benefits in terms of increased throughput, increased CPU availability and reduced jitter. We are not suggesting that kernel-level streaming is the ideal mechanism to manage the resources of a multimedia workstation. Nor do we casually make the suggestion that application functionality be embedded in the kernel. Rather, we view kernel-level streaming as a pragmatic approach to improving support for continuous media within the constraints of a conventional operating system.

## 8. Acknowledgements

We gratefully acknowledge the contribution of our colleagues at the Rutherford Appleton Laboratory, UK, who designed and built the ATM network adapter on which our experiments were based. We would also like to thank the various reviewers who commented on earlier versions of this paper. The work was conducted at the University of Buckingham, UK, as part of the CHARISMA project and funded by the European RACE II research programme (project number R2071).

## 9. References

[1] D Banks and M Prudence, "A High-Performance Network Architecture for a PA-RISC Workstation", *IEEE Journal On Selected Areas in Communications*, Vol 11, No 2, February 1993.

[2] DD Clark and DL Tennenhouse, "Architectural Considerations for a New Generation of Protocols", *Proc ACM SIGCOMM '90*, Philadelphia, USA, September 1990.

[3] C Dalton, G Watson, D Banks, C Calamvokis, A Edwards and J Lumley, "Afterburner", *IEEE Network*, Vol 7, No 4, pp 36-43, July 1993.

[4] P Druschel, M Abbot, M Pagels, L Peterson, "Network Subsystem Design", *IEEE Network*, Vol 7, No 4, pp 8-17, July 1993.

[5] P Druschel and L Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", *Proc 14th Symposium on Operating System Principles*, 1993.

[6] P Druschel, L Peterson, B Davie, "Experience with a High-Speed Network Adaptor: A Software Perspective", *Proc ACM SIGCOMM '94*, London, England, September 1994.

[7] A Edwards, G Watson, J Lumley, D Banks, C Calamvokis, C Dalton, "User-space Protocols Give High Performance to Applications on a Low-cost Gb/s LAN", *Proc ACM SIGCOMM '94*, London, England, September 1994.

[8] K Fall and J Pasquale, "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability", *Proc USENIX Winter Technical Conference*, San Diego, CA, USA, January 1993.

[9] Newton Faller, "Measuring the Latency Time of Real-Time Unix-like Operating Systems", `ftp://icsi.berkeley.edu/pub/techreports/1992/tr-92-037.ps.Z`, 1992.

[10] Berny Goodheart and James Cox, *The Magic Garden Explained: The Internals of UNIX System V Release 4*, Prentice Hall, ISBN 013-098138-9, 1994

[11] JL Hennessy and DA Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc, ISBN 1-55860-188-0, 1990.

[12] NC Hutchinson and LL Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering*, Vol 17, No 1, January 1991.

[13] BJ Murphy, CJ Adams, S Zeadally, "The CHARISMA ATM Host Interface", *Proc 3rd International Conference on Broadband Islands*, Hamburg, Germany, June 1994. Also available electronically as `ftp://ftp.cl.cam.ac.uk/users/bjm22/bris94.ps.gz`.

[14] JC Mogul and A Borg, "The Effect of Context Switches on Cache Performance", *Fourth International Conference on Architectural Support for Programming Languagues and Operating Systems*, April 1991.

[15] J Nieh, JG Hanko, J Duane Northcutt and GA Wall, "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications", *Proc 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, Lancaster, UK, 1993.

[16] MA Pagels, P Druschel and LL Peterson, "Cache and TLB Effectiveness in the Processing of Network Data", Technical Report TR 93-4, Dept of Computer Science, University of Arizona, Tucson, USA, 1993.

[17] C Partridge, *Gigabit Networking*, Addison-Wesley Professional Computing Series, ISBN 0-201-56333-9, 1993.

[18] J Pasquale, "I/O System Design for Intensive Multimedia I/O", *Proc IEEE Workshop on Workstation Operating Systems*, Key Biscayne, FL, USA, April 1992.

[19] Stephen A Rago, *UNIX System V Network Programming*, Addison-Wesley Professional Computing Series, ISBN 0-201-56318-5, 1993.

[20] Curt Schimmel, *UNIX Systems for Modern Architectures*, Addison-Wesley Professional Computing Series, ISBN 0-201-63338-8, 1994.

[21] JM Smith and CBS Traw, "Giving Applications Access to Gb/s Networking", *IEEE Network*, Vol 7, No 4, pp 44-52, July 1993.

[22] DL Tennenhouse, "Layered Multiplexing Considered Harmful", *Proc IFIP Workshop on Protocols for High-Speed Networks*, Zurich, Switzerland, May 1989.

[23] C Thekkath, T Nguyen, E Moy, E Lazowska, "Implementing Network Protocols at User Level", *Proc ACM SIGCOMM '93*, San Francisco, USA, September 1993.

[24] UNIX International OSI Special Interest Group, *Data Link Provider Interface Specification*, UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054, USA, Revision 2.0.0, August 1991.

## Author Information

**Brendan Murphy** is a post-doctoral Research Associate at the University of Cambridge currently working on the integration of continuous media "streams" into a CORBA environment. His other research interests include network protocol design and operating system support for high-speed networks.

**Sherali Zeadally** is finishing his PhD in Computer Science at the University of Buckingham. His research interests include operating system support for multimedia, I/O sub-system design,

especially to support high-speed networks, and distributed systems.

**Chris Adams** is Professor of Computer Science at the University of Buckingham and a member of the Advanced Communications Unit at the Rutherford Appleton Laboratory. His research interests range from global network strategies to hacking processor microcode. They include satellite communications, multimedia applications, multiservice networks and operating systems.

The authors can be reached via electronic mail to `charisma@buck.ac.uk`. An electronic version of this paper is available as `ftp://ftp.cl.cam.ac.uk/users/bjm22/usenix96.ps.gz`. Further information on the CHARISMA project can be found via `http://www.brookes.ac.uk/cms/research/dsproj.html#CHARISMA`.