Writing an Efficient Device Driver for a Multimedia Teleconferencing System*

Alexander Sarris, Satish K. Tripathi
Mobile Computing and Multimedia Laboratory
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742
{*asarris, tripathi*}*@cs.umd.edu*

Abstract

Modern high speed networks, such as ATM, can provide the bandwidth and the QoS guarantees to demanding real-time multimedia applications. However, overall performance of a networked multimedia application will greatly depend on the in-host data movement. Analyzing the characteristics and requirements of those applications, we came to several conclusions about the operation of the multimedia devices' drivers. We applied these conclusions in the design and implementation of a device driver for a multimedia teleconferencing system, based on IBM RS/6000 servers, running the AIX 3.2 operating system. Tracing the complete in-host data path, we found that though our device driver minimized the movement of data between the teleconferencing card and user main memory, the UDP/IP stack proved to be a cause of delay in the movement of data between user main memory and the network interface.

1 Introduction

Networked real-time multimedia applications, such as teleconferencing, are gaining much support and attention, due to the opportunities they offer as versatile communication tools. As versatile and useful as they are, they also produce great amounts of data that need to be transfered among hosts with strict quality of service requirements. Up until the near past, overall performance of these multimedia applications was mainly affected by the inadequacy and best-effort nature of the underlying networks. Especially in wide-area applications, the network was the major bottleneck, and the main candidate for performance improvement. Over

*This work is supported in part by an IBM equipment grant

the last few years, a new network technology has evolved, that promises to solve all network-related problems. ATM networks provide not only the necessary bandwidth, but also the mechanisms to negotiate and guarantee a requested quality of service. Having such a network at our disposal, we used it to support a multimedia teleconferencing system. However, we did not get the improvement in performance that we had expected over the older network technologies. We decided to look for the bottleneck in the in-host data path, analyze the class of networked real-time multimedia applications and search for new optimizations.

Analysis of the characteristics of networked real-time multimedia applications showed that the device driver for the supporting multimedia device can be designed is such a way as to optimize in-host data transfers. With this analysis in mind, we designed and implemented a device driver for a prototype multimedia teleconferencing adaptor. The device driver operation was designed to be uncomplicated, minimized and aggressive. Profiling showed that data was moved in excess of 10Mbytes/sec for the maximum packet size of the teleconferencing adaptor's data packets.

We also traced the data path between the host and the network interface. Profiling revealed that the the UDP/IP protocol stack is responsible for delays in the data path, due to both protocol processing overheads, and data replication.

The paper is organized as follows. In section 2 we do an analysis of networked real-time multimedia applications, and propose the device driver design. In section 3 we describe our experimental setup and our implementation. In section 4 we show our performance measurements of the optimized device driver, and in section 5 the measurements of the UDP/IP protocol stack. We conclude with section 6.

2 Analysis and Design

2.1 NRTMAs

Device drivers are the part of the kernel responsible for direct handling of I/O devices. They provide a level of abstraction to all user processes that need to access an I/O device. The device drivers' efficiency, consequently, directly affects the performance of any I/O-intensive application. One such class of applications is *Networked Real Time Multimedia Applications* (NRTMAs). An example of a NRTMA is real-time multimedia teleconferencing.

Analyzing the characteristics and requirements of NRTMAs, we came to several conclusions about the design of the device drivers for the multimedia devices, as well as for the requirements on the network side of the in-host data path. The obvious characteristic is that the applications use a **network**. In NRTMAs, we really have a number of multimedia adaptors communicating with each other, by means of a network. Inside each host, this translates to the multimedia adaptor exchanging data with the network interface. The in-host data path consists of two parts, the one between the multimedia adaptor and the user application, and the one between the application and the network interface. Overall performance is affected by both of these paths.

The other main characteristic of NRTMAs is their **continuous real-time** nature. Data is being captured, encoded and transmitted[1] constantly, throughout a session. The devices present (and are capable of handling) data in small sized packets, in a constant rate. All data is time-critical must be delivered to its final destination within strict time constraints. Because of this, user-level software is expected to continuously issue requests for I/O, to both the multimedia and the network devices.

Partly because of their continuous real time nature, some NRTMAs move **large volumes** of data. For a movie-quality, 30 frame/sec video teleconference, NRTMAs transfer data loads in the order of Mbits/sec.

One other characteristic of NRTMAs is that they are **single-user** applications. One user may be in a teleconferencing session with a number of other participants, but there is only one user per machine who is capable of using the multimedia resources of the host. There can certainly be multiple multimedia adaptors on the same host for use by different users, and since the network interfaces are shared resources, there is the possibility of multiple NRTMAs being hosted by the same machine. Most, if not all, instances of NRTMAs, however, are single-user applications. With the personal computer becoming more inexpensive and widespread, and the cost of multimedia hardware dropping constantly, it is very unlikely that this situation will change. We can can safely make the assumption, then, that there will be one user per NRTMA per host.

Finally, NRTMAs are **symmetric**. We expect outgoing and incoming traffic to have the same general characteristics. In terms of data encoding, this is actually a requirement, if the multimedia hardware on the ends of the network is to communicate with each other. In terms of data size and data stream characteristics, matters are more complicated. Though outgoing traffic depends solely on the sender, incoming traffic will depend on the number and nature of all the host-to-host links, as well as the existence and utilization of some multicasting mechanism. We are expecting, however, a constant stream of incoming time-critical data, similar in nature as the outgoing stream. We are assuming a reliable network carrier that can provide pre-negotiated guarantees, so congestion and packet loss do not hinder performance, and do not interfere with data flow.

2.2 Device Driver Design

The above observations translate to rather strict requirements for the design of the supporting device driver. We have large amounts of time-critical data, presented in small data packets, that need be moved between the application and the multimedia device. Fast movement of data between the multimedia card and the user application is a requirement that needs no explanation. The real-time nature of the applications imposes the additional, very important requirement, of minimal response time. This includes pure packet transfer time minimization, as well as minimization of processing time, and user process waiting time. Processing time is an important factor, since the data sizes to be transfered are small, and the operation is performed continuously. To summarize, the device driver must be able, upon receipt of an I/O request from the user application, to move data fast, and with the minimum amount of overhead possible.

[1] and also received, decoded and presented

There are certain conclusions we can draw from our analysis, that could allow the above requirements to be met. Since the device driver is interacting with only one process, it need not include any multiplexing capabilities.

In addition, since data will be sent over a network to a similar device, there is no need for data processing. The device on the other end will be able to handle the data itself. Also, as was mentioned above, since the data produced by the device are time-critical, they need to be serviced as fast as possible. The user level software will be issuing read requests constantly, for the duration of the session. Consequently, data produced by the device will not remain unwanted for any significant amount of time. The same applies to data being sent by the application to the device; the device will be able to accept and process them in real time. Data caching, finally, is useless with this class of applications. For the reasons above, it can be concluded that there is no need for data to be buffered inside the kernel at any point during the in-host transfer.

There is also no need for I/O request queuing. As we mentioned above, I/O requests will be always accommodated synchronously by the device. I/O queues are used because of the asynchronous nature of some, mainly block, I/O devices, and to provide a fair distribution of the device to all the processes that are in need of it. Here we do not have an asynchronous device, and there is only one process that can use it. Note, that by I/O queuing we refer to the queuing of the data to be processed along with the request description, as well as pure request-only queuing, like the one used by raw block device drivers. The later do not store data in the I/O queue, but simply the request to move data. Avoiding I/O queuing, means that we avoid all the overhead associated with it, such as creating and managing queue items.

The above two paragraphs call for a device driver that does no buffering whatsoever, and moves data directly between user space and device buffers. They also suggest a driver whose two halves do not communicate and cooperate for I/O processing. We are asking the driver, essentially, to move data upon receipt of a user application request. The top part becomes completely responsible for the accommodation of the request, in real-time. We still, however, need to account for data unavailability or busy buffers at the time of the request. We propose a simple bookkeeping function that can be supported by the interrupt handler, the bottom part of the device driver. The interrupt handler is usually called by the device itself, by means of an interrupt. An interrupt is issued by the device to signify the existence of new data on the outgoing buffers, or to acknowledge the successful completion of a write request. Both interrupts signify that the device is available to accommodate another I/O request. In our approach, upon receipt of one such interrupt, the interrupt handler identifies it, and sends a signal to the user application. The user application signal handler then sets a global variable that corresponds to the state of the device. This approach, as well as an alternative one, are explained in the implementation section. The result of this approach is that there is no need to block the requesting user process, and therefore can keep the device driver design simple and aggressive. The overall design described above minimizes response time, and since all buffering is eliminated, also minimizes pure transfer time. The device driver processing time for the transfers is nonexistent.

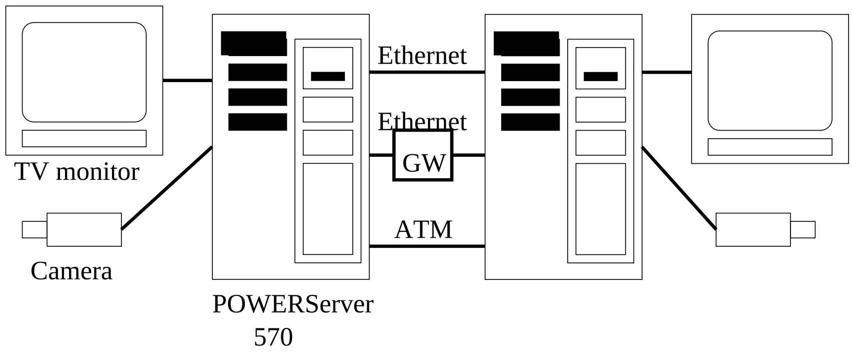

Figure 1: Experimental Testbed

3 Setup And Implementation

3.1 Setup

The testbed consists of two MMT[2] high-end prototype cards, hosted by two IBM POWERServer 570 RS/6000 systems. Each server has a 50MHz POWER processor, and 32 Megabytes of main memory. The bus is based on IBM's Microchannel Architecture (MCA), providing 100MB/sec in non-streaming mode. Data is carried on the MCA bus in strides of 8, 16 and 32 bits, with a 20ns cycle time. Each MMT card is connected to a TV monitor and a camera. The are three network connections between the servers (figure 1):

- The building's Ethernet.
- A back-to-back ATM connection.
- A back-to-back Ethernet connection, via the MMT card's on-board Ethernet interface.

When using the building's Ethernet, traffic goes through a router, denoted "GW" in figure 1. Teleconferencing can take place over any of the above connections. For the first two connections, user level software sets up a UDP/IP connection between the hosts, which is then used to carry the teleconferencing traffic. We run IP over ATM, so the socket abstraction is provided for the ATM interface. The back-to-back Ethernet connection bypasses the host altogether, using the card's on-board Ethernet interface.

3.2 The MCA Bus

The general layout of the MCA bus structure can be seen in figure 2(a). The CPU, like in most systems, has direct access to main memory. Access to the I/O devices is through the I/O Channel Controller (IOCC). The controller is responsible for load and store instructions, interrupts and general bus control (DMA, data buffering, etc.)

Each adaptor on the MCA bus provides addressable resources. In the RS/6000 machines, all adaptor memory resources, such as registers and on-board memory, can be mapped in virtual memory. The RS/6000's, in other words, utilize memory mapped I/O.

[2] MMT stands for Multimedia Multiparty Teleconferencing

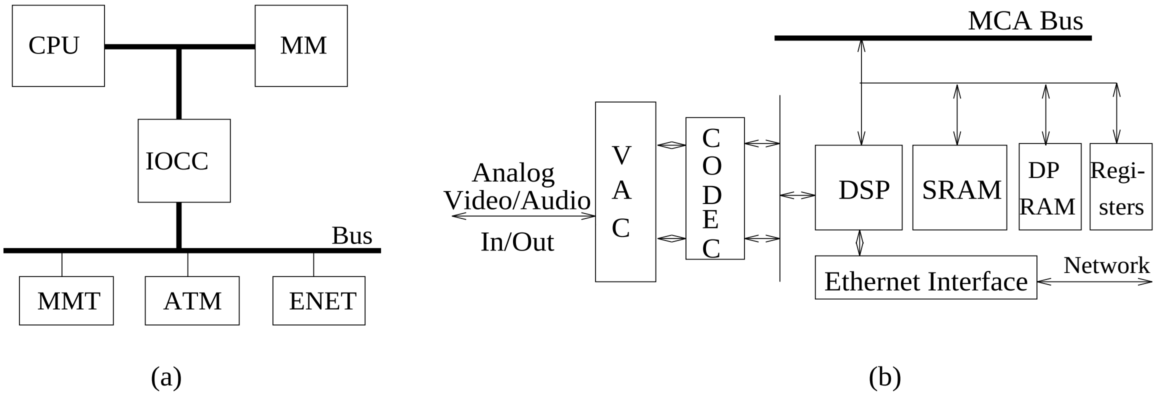

Figure 2: (a) The MCA Bus (b) The MMT Card

3.3 The MMT Card

The MMT system consists of three major parts, the VAC, the CODEC and the DSP[3] subsystems 2(b).

The VAC subsystem is responsible for the conversion between analog and digital video and audio. It converts NTSC video to YUV422 digital video, analog audio[4] to PCM/ADPCM digital audio, and vice versa. Digital audio and video is communicated between the VAC and CODEC subsystems.

The CODEC subsystem processes 64Kbps PCM or 32Kbps ADPCM digital audio. It simply converts 8-bit parallel input to serial output, and vice versa. The part of the CODEC that processes video consists of two engines, one for compression and one for decompression. The engines support the Motion JPEG standard, and can provide a maximum frame rate of 30 frames/sec. Resolution and frame rate are both configurable, and can be different for the two engines. The rate of the produced data stream -controlled by the DSP- is also adjustable, and can range from 128Kbps to 10Mbps.

The DSP is responsible for the management of the whole system. It can support multiple teleconferencing connections (a maximum of 32), and present the incoming video data in separate windows. There are 256 KB of Static RAM (SRAM) and 16 KB of Dual Port RAM (DPRAM) available to the DSP. The SRAM is used by the DSP for the mixing of video and audio, as well as for the smoothing of incoming traffic The DPRAM is used to communicate data to the device driver. There are also several control and status registers that the device driver sets and checks, respectively, to control the operation of the adaptor.

There is also an on-board 10Base-T Ethernet interface. The DSP has direct access to the Ethernet chip, and so data can be exchanged between the network and the CODEC without bus involvement. Once the microcode is downloaded on the card, and a connection is setup, teleconferencing runs independently, and the card uses no more of the system resources.

[3]VAC stands for Video/Audio Capture, CODEC stands for Compression/Decompression, and DSP stands for Digital Signal Processor.

[4]Microphone, or line audio

3.4 Implementation

Based on our analysis, we implemented the device driver for the MMT adaptor. A previous version of the device driver, utilized a service of the bus device (/dev/bus0) and operated similar to a character-based driver. Performance was variant, and rather low for the larger MMT data packet sizes. There also was the supporting user-level software, with which the new MMT driver remained compatible.

The configuration routine registers the driver with the kernel, and perform self-checking and self-configuration. The open() routine checks if the device is available, and updates the local data structures to reflect the current device status. The close() routine clears the busy flags, and makes the device available for opening.. The device driver supports several control and status I/O control (ioctl) options. Upon opening the device, an internal ioctl() call registers the calling user process, so that control signals can later be communicated from the interrupt handler. Other ioctl options return the status of the device components, or set command registers.

The read() and write() routines are symmetric in their way of operation. They are responsible for the complete data transfers. Upon receipt of a user request, they make the necessary processing, and directly transfer data from the device buffers to user space buffers. This is in the spirit of simplicity and aggressiveness analyzed in the design section. The bottom half of the driver does not deal with data transfers at all. Our driver's operation resembles that of the raw character drivers for block I/O devices, with the exception of request queues, and of course, data alignment and random access.

We shall use the read() routine to illustrate both calls' operation.

The dual port memory on the MMT card that is used to communicate data packets is split in two parts: the inbox and the outbox. The read() routine copies data out of the inbox. The operation of the routine is split into three steps:

- The inbox memory is mapped to virtual memory belonging to the kernel.
- Data is moved directly from the inbox to user space buffers.
- The inbox memory is detached from virtual memory.

To attach the inbox memory to kernel virtual memory, the `BUSMEM_ATT()` system macro is used. `BUSMEM_ATT()` accepts two parameters: one is the bus id of the bus where the device resides[5] and the other is an offset in the allocated memory segment, where the memory resources of the device start. `BUSMEM_ATT()` then allocates a new segment for the device, locates the device memory resources' starting address in the segment, and then returns a 32-bit virtual address that can be used to access the device's memory. From this point on, all memory resources on the device appear as virtual memory, belonging to the kernel.

The size of the MMT packets is variable, so the size of the data on the buffer has to be acquired next. The application is always allocating enough buffers for the maximum packet size, and is requesting that many data with the read() system call. The device driver, of course, might return less. To move the data, the uiomove() routine is used. The parameters

[5]The RS/6000s can support multiple buses

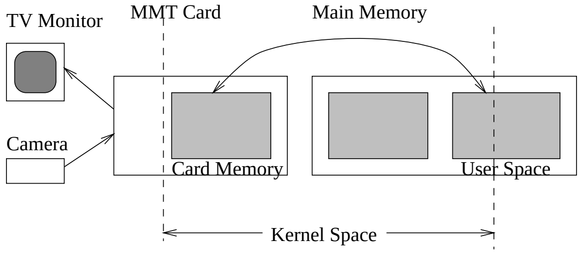

Figure 3: Optimized MMT Device Driver Operation

are: `uiomove(buf_addr, size, UIO_READ, uiop)`. The address of the device inbox as well as the size of the data residing there is specified with the `buf_addr` and `size` parameters, respectively. `UIO_READ` specifies the direction of the data movement. In the write() routine, `UIO_WRITE` is specified. `uiop` is a pointer to the uio structure which was constructed upon receipt of the user application read() system call. The uio structure contains, among others, information regarding the address of the user space buffers that data will be copied to, and the requested data size. After the completion of the data transfer, the structure is updated. Before the data movement The user application buffers that are involved are marked as kernel resources, and locked. The data movement operation is shown in figure 3.

The interrupt handler of the MMT device driver identifies the type of interrupt, and sends a corresponding signal to the user application. There are two types of interrupts that the handler recognizes. The `SIGUSR1` interrupt is issued by the MMT adaptor when new data has been put on the inbox memory. The `INTOACK` interrupt is issued after the card has successfully processed data on its outbox, and the outbox buffer is now free. Upon receipt of a signal that was caused by a `SIGUSR1` interrupt, the process' signal handler sets the boolean global variable `data_available` to true. The signal signifying a `INTOACK` interrupt causes the signal handler to set another boolean global variable, `buffer_free` also to true. We can see these operations in the rightmost part of figure 4.

Figure 4 shows the flow of control, and the interaction of the device driver with the user level software. The user application continuously reads data from a UDP socket and sends them to the MMT card via a write() call, and continuously reads data from the MMT card to send them through the socket. After receiving data from the socket, the `buffer_free` flag is checked, and if it is set to true, then a write() system call to the MMT device is issued. Data is them moved by the device driver to the MMT adaptor buffers, and the adaptor is interrupted so as to take notice of the new data. A similar procedure is followed when reading data from the adaptor and sending to the socket. The call will be reissued, if its corresponding flag is not true.

The procedure we follow is not the mainstream "sleep-wakeup" mechanism. Other character-based device drivers, including the raw drivers for block devices, use this mechanism together with request queues for doing I/O. As we have shown in the design section, our device driver need not support this way of operation. To deal with the issue of buffer availability, we use the aggressive approach of letting the user process know the status of the device every time the status changes. The application issues a request only when that request can be carried out. This way, we avoid the extra overhead of storing and managing the I/O request. Similar

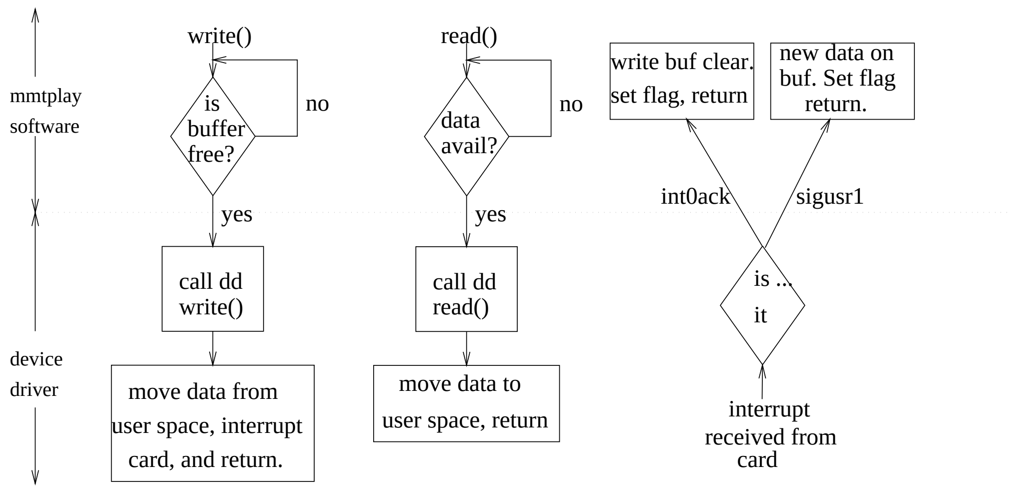

Figure 4: Flow of Control

functionality can be provided by the utilization of select() system calls, which are in fact supported by the MMT device driver. The application would issue a select call inquiring about the status of the inbox or the outbox, before issuing read() or write() call. Our decision not to follow this approach was based on a simple observation. If one of the buffers is busy, then the respective call will not be issued, and another select call will be issued soon thereafter. The context switching involved with the invocation of a select call will then be repeated, until buffers become available. In our approach, context switching occurs only when buffers become available.

4 Results

Our measurements were taken using the RS/6000's microsecond clock. The AIX operating system provides the facilities for easy statistical acquisition, by the insertion of hookwords in the code. The statistics were stored temporarily in kernel space memory, and were later dumped to a file, and filtered. The setting was realistic, with a number of processes running on the system. This way we tried to see if there would be variance in response time. The sample sizes were 1000 system calls, for each of read and write. There are four sizes of data shown in both graphs, namely 532, 1428, 2324 and 3220 bytes. These are the packet sizes of the data produced by the MMT adaptor.

The transfer rates can be seen in figures 5(a) and 6(a), and are presented in MBytes per second. The maximum throughput we obtained was over 10MBytes/sec, for the largest MMT data packet size.

Transfer times are presented in figures 5(b) and 6(b). We are presenting unmodified data, and not averages, so that we can examine the time variance for the data transfers. The dots are the transfer times for the optimized MMT device driver, and the crosses are the times for the previous version of the device driver that had a pure character-based design. The improvement was over 240% for reads, and over 340% for writes. As can be seen in the graphs, the response

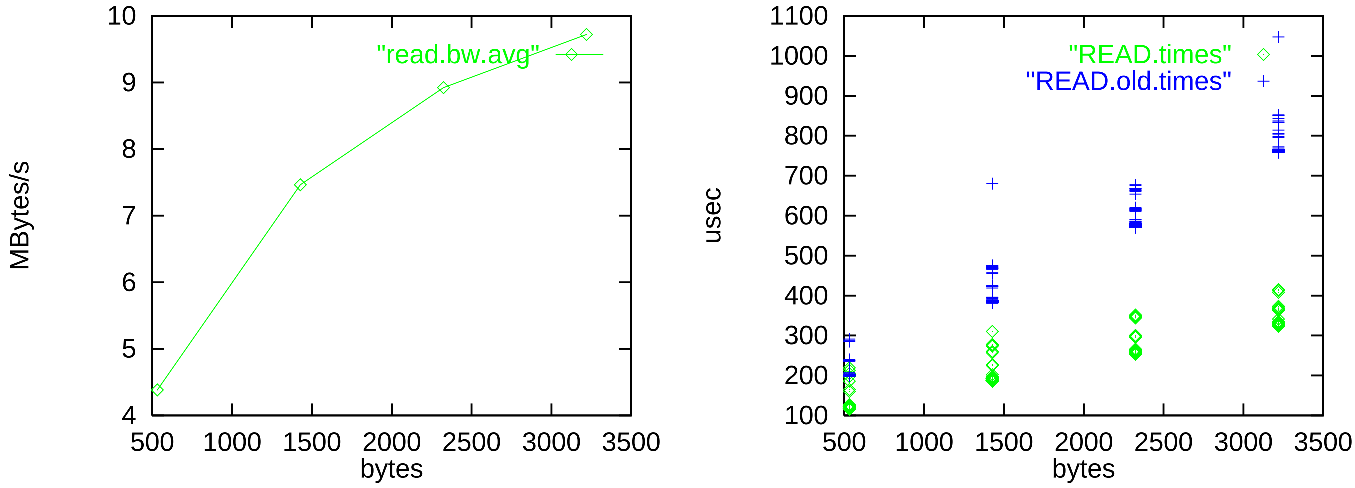

Figure 5: (a) Read Transfer Rates (b) Read Transfer Times

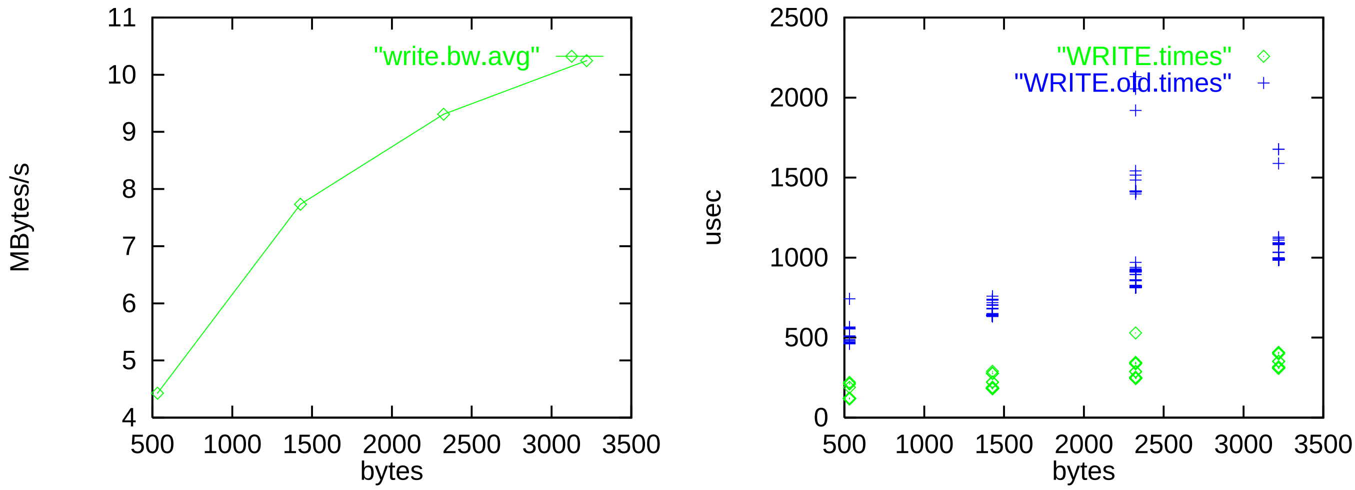

Figure 6: (a) Write Transfer Rates (b) Write Transfer Times

times are relatively invariant for the optimized driver.

5 Sockets

To provide a complete study of the in-host data path, we traced the data path between the application and the network interface. We were motivated to do so mainly because we did not experience the increase in quality that we expected after optimizing the device driver. The method we followed to gather the socket statistics is very similar to the one followed for the device driver. We now had to look at the data in more detail, so as to identify the parts of this more complex data path.

After the application issues a send request, data is received by the socket, and appropriate

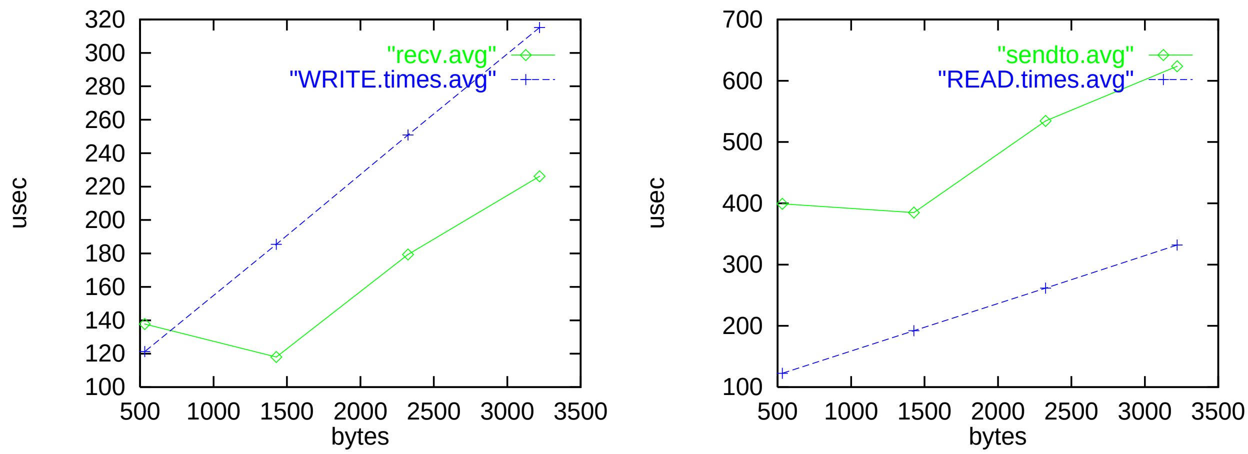

Figure 7: (a) Socket Receive Latency (b) Socket Transmit Latency

packetization and processing is done by the UDP/IP stack. After processing, data resides in an mbuf chain, inside the kernel. A request is then queued to the bottom half of the network adaptor device driver. As soon as buffer space becomes available on the network adaptor, the mbuf contents are copied there[6]. The timing of the last transfer generally depends on available buffers on the network adaptor relative to the size of the data to be transfered and the current status and number of all processes sharing that adaptor (whether or not they have also queued I/O requests for that device). Since the final data transfer, from kernel mbufs to the network adaptor buffers, happens asynchronously to the socket send call, we are not including these times in out plots.

The reverse procedure is followed when data is received on the network adaptor. Data is copied from the network adaptor to kernel mbufs, it is then processed by the UDP/IP stack, and is finally copied to user space. The process of copying data to kernel mbufs and the processing of that data in the UDP/IP stack is also not a direct consequence of the receive socket call. These two steps are executed when the network adaptor device driver receives a "data available" interrupt by the adaptor, and is called to service this request. The receive call is only responsible for the data being copied from the kernel buffers to user space. It is therefore not representing the complete latency of the incoming data path, and neither do the times shown in figure 7(a).

We are including the average transfer times of the MMT device driver for comparison purposes.

As it is obvious from the transmit plot, internal buffering, involving at least one memory-to-memory data copy, along with UDP/IP processing, constitute a system bottleneck. Though the socket abstraction allows for great portability and manages to relieve the applications from network interface concerns, it also causes delays that cannot be tolerated by NRTMAs. There is room and need for new approaches to this problem.

[6]In the case of the ATM interface, data is first moved to a contiguous area in kernel memory, to be later DMAed to the ATM adaptor

6 Conclusion

In this work, we concentrated on the in-host data movement of networked real-time multimedia applications. Analyzing this class of applications, we proposed a framework for the design of the supporting device driver. According to our analysis, this design optimizes data transfers between the multimedia device and the user-level software. We applied this design to the implementation of the device driver for a multimedia teleconferencing card, hosted by RS/6000 systems, running AIX 3.2. Our measurements show that the transfer rates are indeed optimal. We then went on to trace the other part of the in-host data path, that between the user application and the network adaptor. Because of data replication, and UDP/IP processing overheads, we found that part of the data path to be the system bottleneck.

Acknowledgements

We would like to thank Debanjan Saha, for his help throughout this project. We also thank George Apostolopoulos for his help on improving this document. Most of all, we thank Rohit Dube, without whom this report would not have been possible.

References

[1] G. Partridge, "Gigabit networking", Addison-Wesley, Reading, Massachusetts.

[2] Leffler, S. and McKusick, K. and Karels, M. and Quarterman, J. "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison-Wesley Publishing Company, Massachusetts.

[3] MMTplus Funcional Description, IBM Internal Document, May 1993

[4] Technical Reference: Device Drivers, AIX version 3.2 for RISC System/6000

[5] Supporting Distributed Multimedia Applications in ATM networks, Ph.D. Thesis, Department of Computer Science, Univarsity of Maryland, College Park, MD, 1995.

[6] Communications of the ACM, April 1991. Special issue on Digital Multimedia Systems

[7] D.Saha et. al, "A Video Conferencing Testbed on ATM: Design, Implementation and Optimizations", IEEE International Conference on Multimedia Computing and Systems,1995.

[8] C.B.S. Traw and J.M Smith, "A High-Performance Host Interface for ATM Networks", Distributed Systems Laboratory, University of Pennsylvania.

[9] B.S. Davie, "A Host-Network Interface Architecture for ATM", Bell Communications Research, Morristown, NJ.

[10] J.M. Smith and B.S. Traw, "Operating System Support for End-to-End Gbps Networking", Distributed Systems Laboratory, University of Pennsylvania.

[11] D.D. Clark, S. Shenker and L. Zhang, "Supporting Real-Time Applications in Integrated Services Packet Networks: Architecture and Mechanism" , ACM SIGCOMM 1992.