I/O System Design for Intensive Multimedia I/O

Joseph Pasquale

pasquale@cs.ucsd.edu, (619) 534-2673
Computer Systems Laboratory
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

Abstract

The intensive I/O generated by multimedia devices and delivered over fast networks requires a new approach to I/O system software design for workstation operating systems. In general, the system should provide more flexibility in how devices are controlled and in constructing the data paths used for data transfer. I argue for the separation of control and data transfer as a general design principle, and describe how I/O bus bandwidth can be used more efficiently through direct connections between I/O devices.

1. Motivation

Multimedia I/O devices are becoming increasingly available for workstations. Given that audio and video are natural communication media, users want workstations with microphones, video cameras, speakers, CD-ROM drives, possibly a color scanner and printer, as well as the standard keyboard, mouse, and large high-resolution color monitor. The rate of I/O generated by some of these devices is significantly higher, by at least one or even two orders of magnitude, than most devices in use today. For example, an unsophisticated video camera can produce data at rates in the tens of megabytes per second (MBps). And access to intensive I/O devices is not limited to what is connected to a user's workstation, as fast communication networks for workstation computing environments will be able to support data transfers from remote intensive I/O devices.

Given this trend, the role of the workstation operating system as an I/O switcher takes on more importance than ever. It must use I/O resources such as time on the system bus more efficiently to obtain higher throughput, schedule I/O resources to properly obtain real-time response, and provide mechanisms to support synchronization of related I/O streams (e.g. audio and video). While all these issues are important, I will focus on the efficient use of I/O resources, particularly time on the system bus.

2. Problems with Current Models of I/O

Consider the case of HDTV digital video transmitted over a network to a workstation. Packets containing parts of compressed video frames arrive at a workstation's network controller, they are buffered in memory to form a complete compressed video frame, and then frames are sent to the workstation's video controller where decompression takes

place. The compressed data rate is up to 5 MBps using the MPEG compression standard [1].

Although most workstation I/O busses (DEC Turbochannel, Sun SBUS, IBM Microchannel) claim bandwidths of 100 MBps and beyond (based on burst-mode transfers), more typical bandwidths are 10-20 MBps [2]. If the I/O system supports device data transfer based on programmed I/O, the data will be transferred over the bus multiple times, at least 2-4 times depending on the system's CPU and bus architecture. For our video example, this results in a minimum required throughput of 10-20 MBps, a significant fraction of what most workstation I/O busses can provide, and this traffic must compete with other traffic on the bus. (Note that this rate is for a single video stream; simultaneous access to multiple video streams which can lead to higher rates is not an unreasonable expectation). Programmed I/O does have the advantage that the data can be interpreted (e.g. demultiplexing packets to the appropriate network protocol modules, constructing a frame from small packets) as it is being copied, and in a properly designed system it allows the data to be deposited directly in a user process's address space rather than first being buffered by the kernel [3]. The problem is that the CPU spends much of its time simply moving data from one place to another and waiting for memory to be accessed.

If the I/O system supports device data transfer based on DMA, one can get closer to the claimed 100 MBps throughputs between devices and memory. However, CPU-based memory copying still results because it is generally not known at DMA time where to put data if part of it (e.g. a header) is not first interpreted. Also, if DMA cannot be used to move to or from an arbitrary memory location, data may have to first be buffered by the kernel. In effect, DMA can make the throughput problem worse because bus transfers result from DMA operations *plus* CPU-based copying.

Given the high data rates and large volume transfers for video, any extra copying of data has a significant impact on performance. The problem is exacerbated by buffered I/O models where a process reads from the source device by having the data first deposited in a device buffer, then copied to a kernel buffer, then copied to the process's address space, after which the process writes to the destination device by copying to a kernel buffer and finally copying to the device buffer. There may even be additional copies between kernel modules (e.g. layers of network protocol processing). We have built some audio and video applications [4,5] mainly for the purpose of getting first hand experience on where the operating system causes problems, or where it could provide better support. By far, the single-most limiting factor is the time it takes to move around large amounts of data.

This has not been a major problem in the past because most devices are relatively slow, and the ratio of computation to I/O has been relatively high. However, the performance cost of memory copying has never been as high as it is now for RISC-based workstations. For example, the measured throughput when copying large blocks of uncached data on a DECstation 5000, a RISC-based workstation rated at 18 MIPS, running Ultrix 3.1D is 12.6 MBps [6]. And, many applications are becoming purely I/O driven, with little computation involved, as in window-based communication applications, video conferencing where audio and video are retrieved from the network and displayed, or browsing a CD-ROM containing images, video, and audio. This even applies to network routing, where the bulk of the work is simply moving packets from one network device to another. For these applications, it should be possible to transfer data between devices in a light-weight manner.

3. What is Needed

It is interesting that much of the power offered by many workstation I/O busses remains untapped. Most I/O busses like the Sun SBUS, IBM Microchannel, and the DEC Turbochannel support large "burst-mode" transfers. Many support "peer-to-peer" communication, so that data can be transferred directly from one device to another without ever being copied into memory, and without CPU intervention (a noteworthy exception is the Turbochannel) [2]. For large I/O-driven applications, these capabilities can be put to good use.

Some busses even allow for a certain type of preemption, allowing one to consider scheduling the I/O bus similar to the way a CPU is scheduled. This is important when one has to multiplex multiple real-time I/O streams, e.g. video and audio streams, on a single I/O bus so that deadlines for I/O events (e.g. display with the next video frame 33 milliseconds after the display of the previous frame) are met. It is interesting that in the past, operating systems designers have focused on the management of the "traditional" scarce resources, processor time and memory space. In considering intensive multimedia I/O applications, the scarce resource requiring proper management is I/O bus time and/or bandwidth.

Let us first consider hardware, as good I/O performance will require better integration of hardware and software [7]. Devices which involve large I/O should be controllable so that large-grained burst-mode bus transfers to arbitrary memory locations, *even to other devices*, are possible. Devices with these basic capabilities will be referred to as *burst-controllable* devices. Devices which can be programmed to initiate burst-mode bus transfers are even more flexible, and will be referred to as *burst-programmable* devices. The added device functionality I am promoting here is simply the ability to set up data transfers to go directly between devices (as well as between a device and memory), and to control the sizes and timing of these data transfers. Also, I do not preclude the addition of special-purpose functionality on a device, such as on-board compression/decompression for video devices, assuming the function is widely used and can be cheaply incorporated (e.g. by the addition of a single VLSI chip).

The design of these devices involves technology that already exists. Chips for DMA and bus mastering are becoming widely available, especially for the popular workstation busses. Some busses (e.g. IBM microchannel) actually provide a great deal of hardware support for flexible data transfers, eliminating the need to design much of this complexity in the device. The cost for what I propose is easily reclaimed in improved performance for intensive I/O applications.

At the operating system level, the I/O system software architecture should be designed such that *data transfer and I/O control are separated*. By data transfer, I mean the movement of *actual data*, i.e. data actually requested by a consumer from some producer, and not *control data* or "meta-data" such as network headers or file control blocks, which describes properties of the actual data. By I/O control, I mean anything other than data transfer, particularly any operation which provides information about or changes the state of I/O resources, such as determining where data should go by interpreting a header, opening or closing a device, creating or destroying a network connection, allocating or deallocating buffers, and so on. Data transfer and I/O control functions, particularly for intensive I/O systems, have very different characteristics: I/O system software and devices usually need to interpret control data but not actual data; control data is usually

much smaller than actual data; the timing requirements for the delivery of control requests (or control data) and actual data are generally different; the frequency of control requests is usually much higher than data requests, and the distribution of control and data requests over time are very different. MFS, the multi-structured file system described in [8], derived much of its high performance by exploiting these differences.

Separating data transfer and I/O control is a useful design principle with general applicability for large-scale intensive I/O systems. In the case of burst-controllable devices, the CPU does I/O control, setting up devices to communicate data directly between each other. In the case of burst-programmable devices, the devices do I/O control, determining the destination of data, and initiating the transfer. Only if the data is destined for a process does the device transfer actual data to main memory and interrupt the CPU. If the device can determine that it needs to transfer many related blocks of information (e.g. sequence of packets making up a single message), it may collect the blocks and send them all at once, or transfer each block but only interrupt the CPU at the end of all the transfers, or do something in between, all under program control. Having this flexibility is important: to maximize parallelism between CPU and device processors, the programmer must be able to control the unit and amount of data upon which work is done, as too much or too little can reduce performance [9].

Consider a network controller which is simply burst-controllable. When a packet arrives, the header is retrieved (read using individual word transfers by the CPU or a single burst-mode transfer of the header whose size can be effectively estimated or over-estimated if necessary) and interpreted by the CPU. Once the location for the bulk of the actual data is determined, this can be accomplished using a single burst-mode transfer, which may be directed to a device such as a video controller rather than memory. Of course, the receiving device must also be set up for the reception (by way of a control request by the CPU).

We are involved in a project called Sequoia 2000 to address the massive data storage, network, and visualization requirements of Global Change researchers [10]. We are currently experimenting with many of the ideas presented above in our design of the network and I/O software, which must support high-speed communication between a terabyte storage server being designed at UC Berkeley and any number of scientists' workstations located throughout the Sequoia 2000 network [11]. One capability the scientists want is to browse time-sequences of highly detailed images constructed from data collected by satellites or remote sensors, or from data created by climate or ocean models they devise. The data rates needed to transfer these high-resolution images (e.g. in excess of 10 MB per image) which do not compress well (e.g. at most a factor of 2) as a continuous stream for browsing easily exceed the 5 MBps compressed HDTV data rate used in the example above.

4. Conclusions

The intensive I/O requirements for workstations supporting multimedia devices connected by high-speed networks present new challenges in I/O system software design. The throughputs required are significantly greater than for more traditional I/O. Current I/O software simply causes too much movement of data, leading to a system bus bottleneck. What is needed is more direct datapaths between devices, more control over the setup of these datapaths, finer control over the timing of I/O transfers, and separation

of data transfer and I/O control as a general design principle in the I/O system software architecture. This level of control must be provided in some way to programmers, as they know best the requirements of their applications.

5. Acknowledgments

I would like to thank Eric Anderson, Kevin Fall, Jon Kay, Vach Kompella, Keith Muller, and George Polyzos, for all the stimulating discussions and ideas that led to this paper. This work was supported in part by DEC, IBM, NCR, NSF, and TRW. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing the expressed or implied views of the sponsors.

6. References

- [1] D. Le Gall, "MPEG: A video compression standard for multimedia applications," *Communications of the ACM*, vol. 34, no. 4, (April 1991), pp. 46-58.
- [2] T. Dehne, "NI evaluates Sun, IBM, DEC systems: an architectural and performance comparison," *The Sun Observer*, August 1991.
- [3] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, (November 1983), pp. 129-140.
- [4] J. Pasquale, G. Polyzos, E. Anderson, K. Fall, J. Kay, and V. Kompella, "A digital video-conferencing experiment using DECstation 5000 workstations and an FDDI network," in preparation.
- [5] R. Terek and J. Pasquale, "Experiences with audio conferencing using X, Unix, and TCP/IP over an Ethernet," *Proceedings of the USENIX Summer Conference*, (June 1991), pp. 405-418.
- [6] J. K. Ousterhout, "Why aren't operating systems getting faster as fast as hardware," *Proceedings of the USENIX Summer Conference*, (June 1990), pp. 247-256.
- [7] A. Forin, D. Golub, and B. N. Bershad, "An I/O system for Mach 3.0," Technical Report CMU-CS-91-191, Carnegie Mellon University, Pittsburgh, PA, October 1991.
- [8] K. Muller and J. Pasquale, "A high-performance multi-structured file system design," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, (October 1991), pp. 56-67.
- [9] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," *Proceedings of ACM SIGCOMM*, (September 1990), pp. 200-208.
- [10] M. Stonebraker and J. Dozier, "Sequoia 2000: Large capacity object servers to support global change research," Sequoia 2000 Technical Report #1, U. C. Berkeley, July 1991.
- [11] D. Ferrari, J. Pasquale, and G. Polyzos, "Network issues for Sequoia 2000," *Proceedings of IEEE COMPCON*, (February 1992), pp. 401-406.