System Software and Hardware Support Considerations for Digital Video and Audio Computing

Joseph Pasquale

Computer Systems Laboratory
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

Abstract

Digital video and audio (DVA) I/O presents special problems to a computer system's underlying support software and hardware. These problems are due to the high data rates and timing constraints of DVA. To address these problems, more efficient data movement (minimizing data copying wherever possible), and better control over low-level timing of I/O transfers, is required. System software must take better advantage of I/O bus functionality such as burst-mode transfers and peer-to-peer communication. While complex embedded systems are discouraged, controllers may be equipped with special-purpose functionality which causes a reduction in data movement, such as compression/decompression hardware. Finally, system I/O software should support separation of data transfer and I/O control.

1.0 Introduction

Digital video and audio (DVA) computing is the integration of digital video and audio in the user's computing environment. We focus on DVA computing because of the intensive demands it makes on the underlying hardware and software I/O system. This is important, as DVA computing will become a major component of computer workloads. From the personal computer supporting DVA presentation, to workstations supporting DVA acquisition, communication, and presentation, to supercomputers supporting DVA enhancement (e.g. image and sound processing) and compute-intensive compression schemes, DVA applications will be found in all computing environments.

DVA devices for personal computers and workstations are becoming available now due to the lowered cost of a number of technologies which are able to support the massive data requirements of DVA. These technologies

include high resolution color displays for video, digital signal processing VLSI chips to process DVA, optical disks to store the large amounts of data which is characteristic of DVA, compression/decompression VLSI chips to reduce their data sizes, and large memories to buffer this data for transfer to and from DVA devices, to name a few. As fast optic fiber local area networks (e.g. FDDI) are installed, distributed DVA computing becomes possible, allowing resource sharing which is important when considering the large storage and computing resources needed for advanced DVA computing, and allowing workstation-based DVA communication for computer-supported collaborative work. As distributed DVA applications such as video-conferencing and on-demand video services become popular, DVA communication will likely become the major component of traffic on most networks.

Unfortunately, operating system I/O software has not kept up with these developments. Specifically, current operating systems such as UNIX were not designed to support the high data rates of DVA devices, one major reason being the multiple times data is copied from one place to another. Since DVA data is so large, copying generates significant work for the I/O and memory system. Operating system design must shift its traditional focus from processor and memory management to the management of I/O resources (such as time on the I/O bus); or at least, more emphasis must be placed on the latter. Indeed, I/O switching becomes a dominant function for the operating system in supporting DVA applications.

In this paper, we explore what generic functionality the I/O bus, device controllers, and system software, should provide to adequately support DVA applications executing on modern workstations connected by fast local area networks. In Sections 2 and 3 we describe the basic problems one encounters in existing systems, the data copying and low-level control over timing, respectively. In Sec-

tions 4, 5, and 6, we suggest what functionality could be used to advantage in the I/O bus, device controllers, and system I/O software, respectively. Finally, in Section 7, we present conclusions.

2.0 The Data Copy Problem

The reason why DVA I/O is so different from other I/O is because of its high data rate. Telephone-quality audio requires 8 kilobytes/sec (KBps), CD-quality audio requires 175 KBps, NTSC quality video requires 25 megabytes/sec (MBps), and HDTV (High Definition Television) requires 200 MBps. These are sustained data rates (and not just peak), but without compression. Compression will reduce these numbers by a factor of 10-100; however, for video, even after compression the data rates are still a significant fraction of bus and network bandwidths.

Furthermore, users continually want more in terms of quality (e.g. higher resolutions) and quantity (e.g. more video windows and audio channels). Going beyond the standard workstation's large color monitor and speaker(s), these I/O requirements increase rapidly when one begins to consider virtual reality applications. The point is, we are witnessing a dramatic rise in I/O data rate requirements for user interface devices, and this problem will only continue to get worse.

The "data copy problem" is to reduce the overhead in time spent copying data from one place to another within a computer system. Because of the large sizes of DVA data, any data copying that can be eliminated can result in performance improvements.

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 [2-4]. 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 [5]. 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 results in 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 [6]. 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 (direct memory access), 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). In various I/O and network experiments we have undertaken at UCSD, by far, the single-most limiting factor is the time it takes to move around large amounts of data [7-9]. Pasička *et al* have made similar observations when measuring a distributed audio application [10].

The data copy problem 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 accessing memory has never been as high as it is now for RISC-based workstations, which achieve performance by relying on keeping often-used data in registers and on effective caching to keep the processor busy. However, DVA data is generally accessed sequentially; there is no time locality of access (although there is space locality). Furthermore, many applications are becoming purely I/O driven, with little computation involved, as in window-based communication applications, or 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.

There are many reasons for copying data. The reasons of most relevance here have to do with making data accessible to some entity. For example, data located on a hardware device may not be directly accessible by the CPU, and so would have to be copied from the device to memory. Or, if one process has data which another process wishes to access, and they have separate independent address spaces, the data must be copied. Or, where there are different domains of protection as in kernel and user space, data must be copied. Sometimes, data is copied to make future accesses more efficient, such as when data is moved to align its beginning location on a page boundary to allow virtual memory remapping. (Of course, there are other reasons for data copying other than accessibility, such as maintaining control over a version of the data, or copying it to restructure data such as the reorganization of parts or the entirety of the data in data structures.)

Some of these copies can be removed, such as the two copies from kernel to user space and from user to kernel space, when a process assists in transferring data between devices by reading from one and writing to the other. This can lead to significant performance improvements when the I/O data rate is high [10]. In [8], we investigate a general mechanism to allow data to flow between devices through the kernel, bypassing the user process (while the experiments applied to disk devices, the technique is generally applicable to all devices).

3.0 The Timing Problem

Another aspect of DVA is that it is time-correlated: the acquisition and presentation of DVA occurs at specific points in time. To simulate continuity, the display of a sequence of video frames or the sound reproduction from a sequence of audio samples must be synchronized with a real time clock so that these data chunks are transferred to the output device at the correct times (called intra-media synchronization). If the video and audio are related (e.g. a movie and its corresponding soundtrack), then the display of video frames and sound reproduction of audio samples must be synchronized with each other (called inter-media synchronization). See [11, 12] for more information on multimedia synchronization.

The “timing problem” is to assure that data gets transferred from one place to another at the proper time(s). The current problem is that control over timing at the device driver level is too coarse [12]. The time that a process writes data to a device, and the time it actually gets transferred to the device, can be separated by delays over which a process has little or no control.

The combination of high data rates and time synchronization places unusual demands on the underlying system software and hardware. The data rate must be sustained with limited variation. In cases where variation is introduced and cannot be controlled (transmission over a wide-area network), mechanisms (e.g. large FIFOs) are required at the receiving end to smoothen these variations so that presentation seems continuous (at the cost of increased delays).

Control over timing at the bus level is required especially in cases where a device has limited buffering, permitting data to be transferred only when it is time to present it. Not only must there be bus level control over when data is transferred, but control over scheduling the bus may be required. In cases where a very large transfer can cause the bus to be busy for a significant amount of time, other I/O transfers can be caused to wait beyond an acceptable delay, degrading the data presentation process.

4.0 I/O Bus Functionality

To address the data copying problem and the timing problem, we now describe what functionality should be provided by the I/O bus, and in the next two sections, by the device controller and by system software.

Most I/O busses support DMA, allowing data to flow directly between system memory and devices, which greatly relieves the CPU of data movement duties which can be time consuming. Unfortunately, there still exist many high data rate devices which do not have (or have partial) support for DMA, such as DEC's FDDI network controller which only has DMA on the receive side, which limits performance [9] (however, we understand that this is being fixed in future versions of the controller). Furthermore, DMA should support transfers between arbitrary addresses (as opposed to constraining addresses in some fashion, such as word or page alignment).

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 DEC Turbochannel [2,3]). For large I/O-driven applications, these capabilities can be effectively utilized to improve performance.

Some busses like the IBM Microchannel [4] support 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.

5.0 Controller Functionality

What functionality should be embedded in the device controller to help solve the data copy problem and the timing problem? Added functionality often implies complexity which implies cost, so it is worth considering what the general arguments are for simple vs. complex controllers. The arguments for simple controllers are that they are cheap and easy to build. Performance continually improves because most of the functionality is provided by the computer system's basic hardware and software (e.g. CPU and system memory), and these tend to improve faster than any particular specialized device controller. Furthermore, this solution is most flexible as the system can easily be reprogrammed. However, if the basic hardware cannot provide adequate performance, then the

device loses functionality (e.g. not being able to perform a time-critical operation). Furthermore, all data must be delivered to system memory, which may be a substantial bottleneck.

The arguments for embedding functionality in controllers go as follows. Functions can be carried out by processors with memory or specially built hardware (e.g. compression, encryption) located directly on the controller board. This, along with minimal dependence on the computer system, can minimize data movement possibly allowing data to flow directly between devices without need for the system CPU to operate on the data. Unfortunately, the more complex the device, the more costly it is. Performance improvements require purchasing new boards, despite improvements in basic hardware (e.g. faster CPUs and memory). Thus, given technology trends and past history, it is not advisable that controllers be embedded with complex systems.

However, at minimum, we believe that 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 are referred to as *burst-programmable* devices. The primary added device functionality we promote is 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.

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 [4].

In the debate on simple vs. complex controllers, we believe there is an intermediate solution permitting the addition of special-purpose functionality on a device. The main criteria for adding such functionality are the following: it should be cheap to incorporate (e.g. by the addition of a single VLSI chip); there should be a reduction in data movement; data transfers should become more flexible; the functionality should be necessary in that it cannot simply be performed by the main CPU (e.g. due to lack of speed), and its importance should be recognized.

Examples of functionality that meet these criteria are the ability to be bus master and the ability to transfer to arbitrary locations (valuable for multimedia composition). There are certain specialized functions which are becoming standardized such as JPEG or MPEG compression/decompression, parameterized checksum calculation, and DES encryption/decryption; system performance can significantly improve by including VLSI chips implementing these functions on controller boards. However, the CPU should still retain control over timing (when data gets transferred) and the location of the data (where data gets transferred).

Finally, we note that having memory on controller boards is very valuable. This allows data to accumulate until the CPU is ready to schedule a data transfer at a convenient time which promotes efficient data transfer. Large buffers are valuable in removing jitter which get introduced by remote transmissions, especially when going through a large wide-area network.

6.0 System Software Functionality

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, we 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, we 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 control functions, particularly for high data rate 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 generally very different. MFS, the multi-structured file system described in [13], 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 [14].

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).

7.0 Current Work

We are involved in a project called Sequoia 2000 to address the massive data storage, network, and visualization requirements of Global Change researchers [15]. We are currently experimenting with many of the ideas presented in this paper 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 [16]. 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. While the project is addressing a number of problems, I/O performance improvement is perhaps the most difficult.

8.0 Conclusions

When supporting applications with DVA I/O components, a number of system software and hardware problems arise, due to the high data rates and timing constraints of DVA. To address these problems, more efficient data movement (minimizing data copying wherever possible), and better control over low-level timing of I/O transfers, is required. System software must take better advantage of I/O bus functionality such as burst-mode transfers and peer-to-peer communication. While complex embedded systems for controllers are discouraged, special-purpose functionality such as compression/decompression hardware, whose primary goal is to reduce data rates, is valuable and worthy of consideration. Finally, system I/O software should support separation of data transfer and I/O control. A process must be able to control the set up and timing of a data transfer, while device controllers and the I/O bus should manage the actual transfer of data

9.0 Acknowledgments

The author is grateful to the National Science Foundation, Digital Equipment Corporation, IBM Corporation, and NCR Corporation, who have supported this work. The ideas in this paper have resulted from vigorous discussions in the Operating Systems Research Group at the UCSD Computer Systems Lab. In particular, the author is indebted to Eric Anderson, Kevin Fall, Jon Kay, Vach Kompella, George Polyzos, and past members of the group. It is fair to say that the level of agreement on the various points made in this paper varies widely.

10.0 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] Digital Equipment Corporation, "DECstation 5000/200 KN02 System Module Functional Specification," Workstation System Engineering, Digital Equipment Corporation, August 1990.
- [4] P. A. Bowlds, *Micro Channel Architecture: Revolution in Personal Computing*, Van Nostrand Reinhold, New York, 1991.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] K. Fall and J. Pasquale, "An Experiment Using In-kernel Data Paths to Improve Disk I/O Throughput and CPU Availability in UNIX," *Proceedings of the USENIX Winter Conference*, (January 1993), to appear.
- [9] J. Kay and J. Pasquale, "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000," *Proceedings of the USENIX Winter Conference*, (January 1993), to appear.
- [10] M. Pasioka, P. Crumley, A. Marks, and A. Infortuna, "Distributed multimedia: how can the necessary data rates be supported?" *Proceedings of the USENIX Summer Conference*, (June 1991), pp. 169-182.
- [11] R. Steinmetz, "Synchronization properties in multimedia systems," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, (April 1990).
- [12] D. C. A. Bulterman and R. van Liere, "Multimedia synchronization and UNIX," *Proceedings 2nd International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Germany, (November 1991).
- [13] 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.
- [14] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," *Proceedings of ACM SIGCOMM*, (September 1990), pp. 200-208.
- [15] 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.
- [16] D. Ferrari, J. Pasquale, and G. Polyzos, "Network issues for Sequoia 2000," *Proceedings of IEEE COMPCON*, (February 1992), pp. 401-406.