Configurable Data Manipulation in an Attached Multiprocessor

Marc F. Pucci Bellcore

ABSTRACT: The ION Data Engine is a multiprocessor tasking system that provides data manipulation services for collections of workstations or other conventional computers. It is a back-end system, connecting to a workstation via the Small Computer Systems Interface (SCSI) disk interface. ION appears to the workstation as a large, high speed disk device, but with user extensible characteristics. By mapping an application's functionality into simple disk read and write accesses, ION achieves a high degree of application portability, while providing enhanced performance via dedicated processors closely positioned to I/O devices and a streamlined tasking system for device control.

The programming model for ION supports the notion of separation of control function from data transmission. Typically, a small list of data manipulation directives is transmitted from the workstation to the ION node, where data filtering or other forms of processing occur. Only results, as opposed to all data, need be returned to the workstation. In the extreme case, the ION system can acquire all input data and generate all output data, without any processing occurring in the workstation. An example application uses a simple set of directives to capture and digitize high quality stereo audio, mix it to monaural, rate adjust the digitized samples to ISDN rates, convert from binary to mulaw encoding, and transmit the result to a workstation.

ION is being used as an experimental platform for voice mail services in a userprogrammable telephone switch prototype, and as a tool for measuring the I/O performance of computer-disk interfaces. Applications under development include an automated camera positioning system and an object repository.

1. Introduction

The workstations that exploit the rapidly advancing state-of-the-art in processor technology can often be a bane to developers of applications that utilize dedicated special purpose hardware or that impose strict access requirements on conventional hardware. Such evolving systems can suffer from:

- Constantly porting hardware dependent components to new hardware.
- Being locked into a particular vendor to avoid major hardware disruptions.
- Forcing the use of high-end stations because entry-level stations are not easily expandable.
- Constantly upgrading local workstation based device drivers to coexist with operating system releases.
- Relying upon an operating system that is not appropriate for the system's functionality.
- Insufficient workstation capacity to support the hardware requirements of the application.

Applications tied to obsolete processor technology will soon suffer from comparative performance problems as newer workstation technology passes it by. However, interfacing new workstations to an existing hardware base is not simple. Initial workstation offerings often possess meager expansion characteristics, typically just a disk and network connection, so achieving even the electrical connection can be difficult.

Ideally, utilizing a new workstation should entail only simple re-compilation of the application code; however, machine dependencies that result from the use of special purpose hardware complicate a code port. Workstation hardware may not be portable to different manufacturer's stations or even across a line of workstations from the same vendor. This can lead to the loss of a significant hardware investment as working components must be redesigned. Supporting multiple versions of hardware in order to preserve customer satisfaction with older configurations can also be expensive. Even hardware common to multiple stations, which is currently possible since many stations now offer VME bus interfaces, may still require device driver changes and must also track operating system variations from release to release.

An additional problem of using special purpose hardware on a conventional workstation is that the internal structure of the host operating system may not be conducive to the requirements of the hardware. It may be preferable to model an application into subtasks, each with its own flow of control; however, the relatively expensive context switch time for a general purpose operating system may make such an implementation infeasible for performance reasons. Also, the data rates generated by some hardware may have a detrimental effect on other functions in the workstation. In general, it is best to place compute power as close as possible to the source of data, passing only results or preprocessed information on to higher levels in the system. In this manner, devices requiring rapid response need not interfere with time-sharing operations.

ION addresses these problems by partitioning an application into hardware dependent and independent components, and providing a vendor independent interface between the two. The hardware independent components reside in the workstation, and are therefore easily ported to new architectures. The hardware dependent components are situated within a separate backplane-based environment, which is portable in its entirety across workstation changes. The low level connection between these components is the Small Computer Systems Interface (SCSI) disk interface, ANSI X3.131. Since each workstation accesses ION using its local disk system, which is a stable, well-defined interface, there is no need to change vendor supplied host sys-

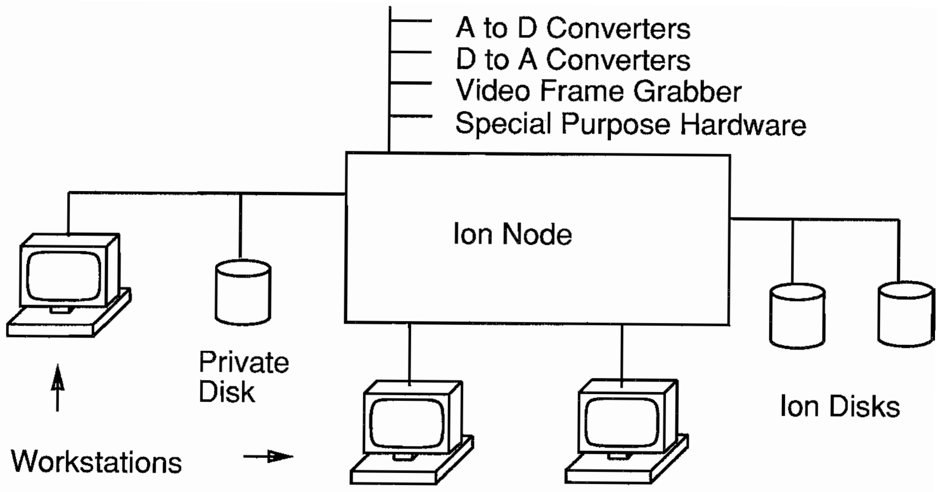


Figure 1. An ION system. Multiple workstations connect to an ION node, which contains single board computers and other peripheral interfaces and devices. Each workstation views its ION connection as though it were a large conventional disk drive.

tem software. Current SCSI performance capabilities also provide a respectable (5 megabyte per second) data access rate.

ION configurations are expandable and sharable as needs dictate. Additional single board computers (SBC's) in a backplane can connect multiple workstations to the same set of hardware resources, or provide extra CPU cycles for I/O devices or applications that require it. Further expansion is possible by using bus repeaters and local area networks to interconnect multiple ION nodes together. The basic structure of an ION system is shown in Figure 1.

2. *The ION Interface*

A workstation sees ION as though it were physically a local disk drive (an ION drive) with a data capacity of 2 terabytes (the SCSI limit). Software running within the ION system mimics the behavior of a conventional device, providing the workstation with a peripheral that it knows how to deal with. The "data" contained in this pseudo-disk device can be random read/write data, traditional file system data, or more complex objects for a variety of applications managed by tasks running within the ION system. The latter is implemented by defining application specific functions, called *actions,* that are enabled by reading or writing specific disk block addresses within the ION drive.

For example, ION supports an analog to digital (A-to-D) conversion application that provides voice messaging service for a prototype telephone switch. The bulk of the application resides in a conventional workstation, while the peripheral devices are located within ION. The application's interface to the A-to-D converters is implemented as an action defined on a set of 5 disk block addresses, each corresponding to 1 of the 5 analog channels. The controlling program within the workstation merely reads from one of these designated disk block addresses to obtain the converted data (**lseek()** followed by **read()** in the Unix domain). By defining such interactions in terms of standard disk read and write accesses, the application remains portable across workstation changes, operating system releases, and to a large degree, complete operating system changes (e.g., Unix to VMS), while preserving any existing special purpose hardware investments.

A further advantage of the disk-like interface of ION is its robustness in the face of application failure. Since ION mimics a local disk drive, the worst case scenario for failure merely results in the apparent symptom that the ION drive has gone into an *off-line* condition equivalent to a real drive losing power or spinning down. This should not have any long lasting effect on the workstation and is remedied by rebooting the ION system.

3. System Architecture

The hardware configuration of an ION node is shown in Figure 2. The current ION configuration uses high speed Motorola 68030 microprocessor based single board computers (SBC's). A port to an Intel 960 based product is underway, although the current system only deals with homogeneous processors. These processors offer sufficient power for the current set of ION I/O devices and will be upgraded to faster processors when more demanding peripherals are in use.

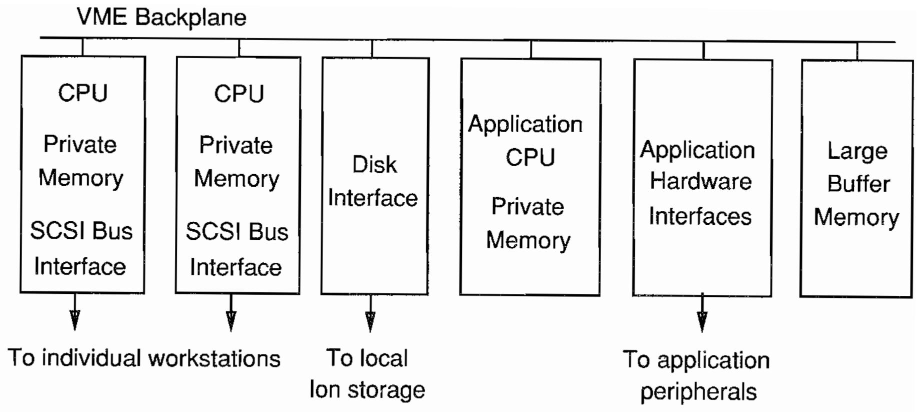


Figure 2. An ION node. Each node contains a dedicated single board computer (SBC) to manage each workstation interface. Other SBC's control local storage, manage object repositories, control additional I/O devices, or run application code.

An SBC is dedicated to each workstation connection, primarily because most hosts insist on using the same SCSI bus address. Each SBC contains its own SCSI interface chip and DMA interface, and is capable of transferring data directly into its private memory without creating external (VME) bus activity. Additional disk interfaces are used to control local node storage, which may consist of file system data or application managed object repositories. Large buffer memory, on the order of hundreds of megabytes, is used as a cache for physical device data. A network interface (Ethernet) connects an ION node to other hosts.

SCSI exploits the use of programmed intelligence within each device on the bus, offloading many functions otherwise performed by the host. It is the fastest common interface supported by a variety of workstations. None of the design issues in ION are constrained to SCSI, and the current version of the system uses Ethernet and serial line interfaces for additional connectivity. Until better interfaces are commonly available, SCSI provides the fastest, common, expansion interface and has a defined next generation architecture offering significantly higher performance (40 megabytes per second SCSI-2). Additional details on SCSI can be found in the appendix.

3.2 Software

ION is implemented as a fast tasking system, specifically designed to support peripheral devices by adding flexible processing power to their functioning, and then interfacing the enhanced device to other conventional computers in an easy and portable manner. Modularity is achieved by dedicating tasks to specific system functions, and passing requests for service through client-server transactions on the same or other ION processors.

All ION tasks are memory resident and execute with their own flow of control. Although they share a common address space, tasks typically do not share any data, relying instead on the fast communications mechanisms available in the system. Most tasks are simple filters that read an input queue, process data, and pass results to an output queue. Tasks are classified into 3 fixed priority groups: interrupt, normal, and background. Within a group, tasks are non-preemptive and run until completion or resource unavailability. Across groups, tasks can be preempted, essentially allowing interrupt tasks to respond to hardware as quickly as possible.

While state machines rather than multiple tasks are often used for managing disk-like operations [2], the recursive state machines necessary for SCSI are difficult to design and enhance. Alternatively, assigning a task to the management of each individual workstation connection and I/O device simplifies the coordination of multiple objects, which in turn allows for easier parallelization of I/O activities. Individual SCSI tasks manage their own disconnect/reconnect behavior on the SCSI bus on a device by device basis. The multiple flows of control offered by the tasking system are useful for application as well as system functions. Such operations as consistency management, network control and routing, and recovery management are more easily designed as separate tasks. Multiple processors fit naturally into such an environment and provide needed power and responsiveness for handling multiple powerful workstations and devices.

While acknowledging that multiple tasks can lead to a loss of performance [3], ION alleviates this by exploiting certain characteristics of its environment: With a single address space and no need for the complete functionality of a general purpose operating system, many optimizations are possible. Task switching, event synchronization and interrupt response time have been designed with minimal overhead. Table 1 summarizes some of these characteristics for the 68030-based system. The cost of using a tasking system over a conventional state machine can be seen in some of the performance measurements presented in Section 6.

Null subroutine time	1μs
Interrupt dispatch time	4μs
Null interrupt service time	9μs
Task switch	25μs
Event synchronization	16μs
Simple system call	8μs
Same processor null client/server interaction	105μs
Remote processor null client/server interaction	140μs

Table 1: ION system characteristic measurements.

Interrupt dispatch time is the delay between when an I/O device signals its need for service and when its interrupt service routine is entered. Null interrupt service time is the time needed to save and restore pre-interrupt state and increment a counter. The task switch time measures the delay incurred when one task suspends and a second task continues execution without any specific form of synchronization. It is mostly the time required to save and restore 2 sets of the general purpose registers of the 68030. Event synchronization time must be added to the basic task switch time when 2 tasks synchronize through the data queueing and dequeueing primitives. A simple system call is similar in timing to the null interrupt time since much of the same functionality must occur.

The null client/server interactions are essentially remote procedure call (RPC) interfaces between cooperating tasks. When on the same processor, this involves task-switching the receiving and sending tasks, queueing and dequeueing the request and response data, and determining the location of the sending and receiving queues. When the RPC crosses processor boundaries, the timing includes the single processor case above plus interrupt latency for sending and receiving, and extra interrupt processing necessary in the processor-to-processor communications functions. (A single interrupt indicates message reception from multiple processors in the system, so a number of input sources must be checked for the presence of a message.) This interprocessor communication takes place over a shared backplane bus, and copying of data can therefore be avoided. It should be noted that the processors are essentially independent of each other they do not share tables or other system data. All interactions occur by placing data in appropriate queues.

The point of the above measurements is not that ION is the fastest system around (it is not), but that a system constructed out of traditional piece parts and written in a high-level language is capable of performance that encourages the application designer to use separate independent tasks for services rather than constructing large complex procedures. The 25:1 ratio between task switch time and null subroutine execution is encouraging, but is still an order of magnitude higher than desired. More favorable is the 3:1 ratio of task switch time compared to a null, non-blocking system call, which indicates that the overhead associated with a request for service is close to becoming independent of how that service is implemented. The 30% overhead of the remote null-RPC over the local null-RPC is also encouraging, and suggests that the location of a service need not be constrained by closeness to its clients.

3.3 Internal ION System Services

The system primitives provided by ION fall into 5 categories:
Task control: Create or destroy a flow of control. Tasks can be created at interrupt time.

SCSI application interface: Define the set of disk block addresses to which an ION application will respond, and the action function to invoke when a workstation reads or writes a block from the set.

SCSI hardware interface: Exchange data with the workstation across the SCSI bus. Also, disconnect and reconnect from the bus to improve bus utilization.

Message exchange and task synchronization: Queue and dequeue messages. This is also the only task synchronization facility available in ION, essentially combining task activation with data availability.

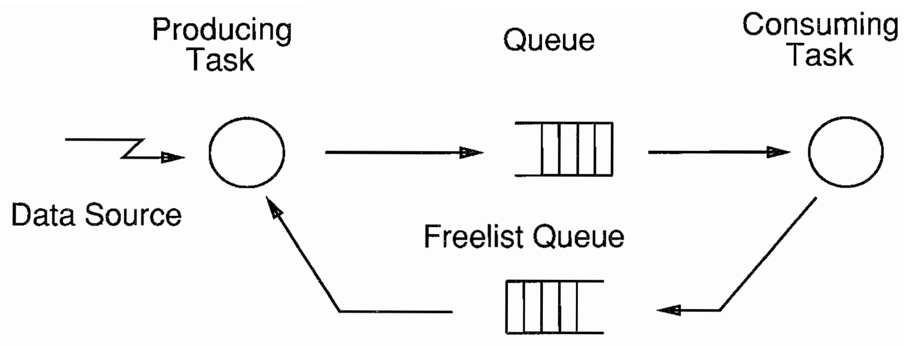


Figure 3. Data queueing model. Queues are used to pass data between tasks, are of infinite length, and never block on a "queue put" operation. Queues are the only mechanism for task synchronization in the system. Data flows in a closed-loop path between tasks to control resource consumption.

Message buffer manipulation: Allocation and duplication of system buffer memory. Three types are available: cached, uncached, and external bus (VME) memory. Cached memory is traditional system memory that can be cached by the processor's memory system hardware for faster access. Uncached memory is used for regions of local memory that can be changed by I/O devices or external bus references. This class of memory is necessary for SBC's that do not have snooping caches. VME memory is the pool of external memory available for buffering large quantities of device data.

3.4 The Data Queueing Model

The queueing of data objects for task-to-task communication uses a straight forward producer / consumer model as shown in Figure 3. The queue structures do not contain space for the pointers to the queued objects, but rather some space is claimed in the object for this purpose. In this manner, queues are effectively of "infinite" length, where any task capable of creating an object is guaranteed to have a place to put it. Tasks never block placing data on a queue, but only by dequeueing from an empty one.

ION places minimal structure on passed data: the data are manipulated by:

- list pointers used for queue linkage,
- a free-list pointer used when a buffer is no longer needed, and

- buffer length, data length, and data offset indicators used for defining the amount of data within the object.

The data description is flexible enough to allow the expansion and contraction of a data object without requiring recopying. The offset and data length describe the location of data within the physical buffer. As an object is passed through a set of tasks, data can be added or removed from either end by adjusting these values within the confines of the actual buffer size. Thus, for example, protocol layers can add wrappers to data without extra copying.

The queue structures contain two optional activation procedures that are invoked during queueing and dequeueing. A *put* routine is called after a data object has been placed on an empty queue. This allows a hardware device, which is not being maintained by a task, to "kick" the device in order to reactivate I/O. The I/O completion interrupt is used to maintain data flow when the device is active. A *get* routine is called before a dequeue operation on an empty queue. This allows a "status queue" to be constructed which will return information about an application as it exists immediately before the dequeue operation.

3.5 Flow Control

Flow control is the mechanism used to control the rate at which data moves about in a system. It prevents resource starvation by limiting the activity of tasks that generate data too quickly for consuming tasks to absorb. The conventional mechanism for flow control uses an upper and lower limit on the number of elements that can be stored in a queue (also known as a high and low-water mark). A process that attempts to queue more than the upper limit will suspend execution. It will be reactivated when consuming tasks reduce the number of elements below the lower limit for the queue. These two values add hysteresis and prevent the thrashing which can occur if a single maximum were imposed on queue size.

Several complications exist with this scheme. Scheduling is impacted because tasks can be suspended at queueing time, rather than just dequeueing time. The ordering of queue operations cannot be guaranteed across separate tasks since it is impossible to predict the closeness of a queue to its high-water mark. The number of buffers in use by a pipe-line of cooperating tasks will grow with the number of tasks. This can stress the stability of the system's buffering mechanism. Finally, tasks must be designed with the understanding that any given queue operation can cause task suspension, rather than the more intuitive approach of suspension occurring only when data are requested.

Because ION tasks do not block on queueing, flow control cannot be implemented in this manner. Instead, closed-loop paths are established that link together the original producer of data to the final consumer as well as all intermediate tasks. The advantage of such a scheme is that the maximum number of buffers that will be used by a particular application is constant and can be preallocated. Also, a runaway task will be throttled back when it waits for a buffer to be released by its last consumer. This scheme also permits trivial tuning of the amount of read-ahead or pre-fetching that an application can perform, since this will be limited by the number of assigned buffers in the pipe-line. Finally, the system does not waste time constantly performing dynamic buffer allocation and deallocation, which can require searching and coalescing of free space. Buffers remain in use until an application terminates, and are then returned to the global memory pool.

3.6 Specifying the Data Manipulation Description

A data manipulation list is used to describe the interconnection of tasks, define buffer allocations, and construct the closed-loop free-list circuits. The syntax is similar to the I/O redirection of Unix, except that the outputs and inputs specify queues, not files, and can indicate multiple inputs or outputs:

```
command argument<input_queue1 input_queue2 ...
     >output_queue1 output_queue2 ...
```

where a queue specification includes:

```
queue_name : free_list ( number_of_buffers, size_of_buffer )
```

The *command* can identify a built-in generic task such as a mixing or duplicating operation, or can specify hardware interfaces, which should generate or accept data from the indicated queues. The *free list* specification is used to supply buffers to a task that must create data, rather than simply modify existing data. The buffering details are op-

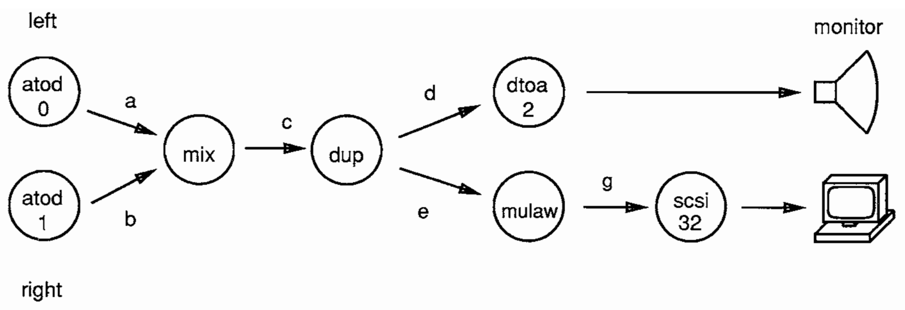
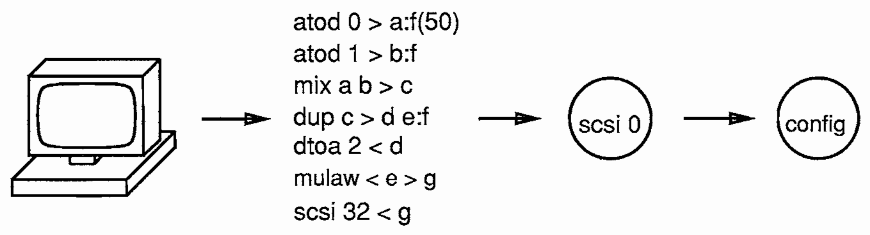


Figure 4. Task connections for an audio mix application. The closed-loop free-list connections are omitted for clarity, but feed into the *atod* and *dup* tasks, and drain from the *dtoa, mix* and *scsi* tasks.

tional, and a default size and number exists for each hardware device.

Figure 4 illustrates a simple example of an application used to mix a stereo source of analog data into a single stream, duplication of the stream for 1) monitoring at a loudspeaker, and 2) mu-law compression and output to a workstation by reading from an indicated SCSI I/O address.

The ION description of this behavior is shown in Figure 5, and is sent in a data buffer to a particular SCSI block address, placing it into a corresponding input queue. A previously created task that awaits input from that queue accepts the list, creates the tasks and buffers, and activates all the I/O devices. Further instructions can be sent to tear down the connections by disabling the *atod* devices. This will cause an end-of-file indication to pass through the system as tasks discover their inputs have closed and in turn close their outputs. Further details on the internal construction of a similar data manipulation can be found in the example application section on analog to digital conversion.



```
atod 0 > a:f(50)
atod 1 > b:f
mix a b > c
dup c > d e:f
dtoa 2 < d
mulaw < e > g
scsi 32 < g
```

Figure 5. Data manipulation description. A short sequence of instructions is prepared by the workstation to configure the data acquisition tasks.

4. Relationship to Other Systems

ION's tasking system is similar in scope to such minimal kernels as Alpha [5], Arts [6], Chaos [7], Mach [8] Ra [9], Spring [10], Synthesis [11], V [12], and others. However, ION is specifically geared towards supporting peripheral devices and then interfacing the resultant modified, intelligent peripheral to other conventional computers in an easy and portable manner. Most of these systems provide similar task manipulation services and are optimized for low latency service requests. Unlike the Spring kernel, ION does not support dynamic deadline guarantees for service [13], but uses a closed-loop control path with a fixed number of preallocated buffers to control service by limiting availability. This scheme prevents a task from using excessive capacity by restricting its rate of production to the rate of consumption, while controlling the amount of read-ahead that can occur in an application. Although ION requires the user to assign fixed scheduling priorities to tasks, which is unnecessary in Spring, the use of 3 levels (interrupt service, normal task service, and background service) and the closed-loop buffer mechanism has proved sufficient for all applications constructed to date.

The programming model of ION resembles a large-grain data flow system [14] such as that used in Max [15]. It is also similar to the Unix Streams package [16] although ION queues are not associated in pairs, and the processing of queue data is done by the Unix equivalent of a process rather than at interrupt time. The information flowing between stream queues is typed to indicate whether it contains data or various forms of control, whereas ION information is untyped.

The notion of sending a configuration description for required data processing to a remote site is similar to the NeFS protocol specification [17] for remote file system access. This system has the advantage of using an interpretor to avoid system dependencies, while ION uses predefined generic services or compiledand-down-loaded modules that necessitate a system reboot. The NeFS programs sent to a remote host are intended to perform file system operations, possibly return data, and then terminate. ION descriptions can be used to establish perpetual data flow, or to set up transactions that are terminated by subsequent requests.

Some of the features used in ION are now appearing in the commercial marketplace. For example, an analog data capture device [18] is available which returns data to a user in a manner similar to that of the voice messaging system described below. However, ION offers the advantage of user programmability of the interfaces and device characteristics. This leads to greater functionality and power located off the host processor and in the peripheral device. ION can also accommodate multiple different devices within the same physical configuration, supporting data filtering operations that can reduce the amount of data that passes through the workstation, thereby improving system efficiency.

5. An Example Application—Analog to Digital Conversion

ION provides the platform for analog to digital (A-to-D) services for a voice messaging application of a prototype programmable telephone switch system called GARDEN. It provides the physical interface to readily available VME cards, and also provides additional processing power to off-load the interrupt handling and data formatting necessary for their operation. It has already provided protection against obsolescence of the hardware investment, since the workstation running the application has already been upgraded, without any impact on the I/O component of the application software. Additionally, since the hardware dependent A-to-D code remains within ION, no driver changes to the host's operating system are necessary upon workstation upgrade.

The part of the A-to-D application that resides within ION is structured around three cooperating tasks. One task is activated by periodic interrupts from the hardware and extracts the raw data from the converter, placing it into a queue for temporary storage. Since the data extraction is not done at interrupt time, less system activity occurs at a high CPU priority level. The interrupt routine and the task share a pair of queues and a token which is passed between the queues to coordinate activity. This prevents the interrupt routine from reactivating the task if the task has not completed its previous data extraction.

The second task is a generic system utility that translates 16-bit linear data into 8-bit mu-law data, as required by this particular application. It is essentially performing data compression on the input stream.

The third task interfaces to the SCSI bus and returns data to the workstation when requested. This task defines a SCSI action function which contains 4 block addresses for each of 5 A-to-D channels. Each channel contains a block address to start conversion, stop conversion, return status, and retrieve A-to-D data.

The part of the application that runs on the workstation requests converted data in response to a start/stop signal from other system hardware, which indicates the beginning and end of a recording session. Upon start, the workstation reads the A-to-D start address for an appropriate channel, activating the device. It then retrieves data by reading the data block address for that channel, while also monitoring for an end-of-session indication. When the latter occurs, the workstation reads the stop address, halting the data conversion. It continues to read the data address until all buffered data have been obtained. The channel is then available for reuse.

5.1 SCSI Flow Control

An underrun condition occurs if the workstation requests data from a channel without any data. At this point, two alternatives exist: The application can suspend the host's I/O operation until data are available, or it can return immediately with some indication that the workstation program should reattempt the data request at a later time. The former approach results in a simpler form of data access, where flow control can be extended into the host system by delaying the completion of I/O operations issued by the host. The latter alternative is essentially polling, which can be inefficient and decrease SCSI bus utilization. However, waiting for the data to be available will tie up the workstation's channel into ION, making it impossible for other applications to communicate over the SCSI bus. (Only the ION connection is affected, other SCSI devices are still accessible.) Hence, if only a single connection to ION is required, the application can be designed to suspend on data availability, while using a polling mechanism if multiple connections are necessary. Both versions can be providing simultaneously by defining 2 action block addresses for data access—one which delays host I/O completion, the other which returns a "poll again" result. The host software would determine which block address to use based on the number of connections in use.

The problem of limited channels can be mitigated somewhat by using 1 of the 8 logical unit numbers (LUN's) defined by SCSI for subdevice access, effectively giving 8 independent channels into ION. However, not all workstations support multiple LUN's. The problem is solved with SCSI-2, which includes a *tagged command* facility that allows multiple outstanding commands to be issued to a single target device. Both the workstation and the target are responsible for remembering that multiple jobs are pending, and properly coordinating the returned information across the SCSI bus.

6. Using ION to Measure Processor-to-Disk I/O Performance

Another application of ION is a measurement tool for studying the SCSI interface performance of a computer system. Contrary to measurement systems such as IOStone [19] and IOBench [20], which use synthetic workloads as a basis, or trace driven studies that require system modifications [21], these measurements are made from the perspective of the disk device, not the workstation, and therefore reflect hardware capabilities, not software characteristics. Also, the procedure used does not require any changes to the host systems. The data thus captured can be used to optimize software performance within the operating system or guide the design of data access routines in a sophisticated user application.

The performance measurement system is defined as an action over a large range of block addresses, corresponding to the "A" section of a standard raw disk device partition table. The action function records the time and type of SCSI state transitions, and the amount of data transferred, and returns or accepts host data immediately. No intermediate SCSI disconnection occurs. From this information, throughput, transfer rate and overhead calculations can be made. Measurements were taken on 3 sample workstations, referred to as Systems A, B and C*. Each workstation is connected to an ION system in an idle envi-

* The workstations employed in this exercise were selected for the sole purpose of illustrating this application of ION. The exercise was not intended as an exhaustive or scientifically precise analysis of computer products. The results reported herein are merely examples of results achieved and should not be considered as either positive or negative judgements about any product or vendor.

ronment. For contrast, comparison to a second ION system is also shown. Unless indicated otherwise, all measurements are taken using instrumented code running only in the ION system and initiated through the raw disk system interfaces provided by each workstation.

The timers used for each measurement are triggered by the interrupts that correspond to phase changes of the SCSI protocol (each SCSI command can be composed of multiple instances of 6 types of information exchanges called phases). Hence, such measurements are not influenced by operating system overhead, which is subject to considerable variation between vendors. The systems were operated in the asynchronous SCSI data transfer mode, as this was the only mode of operation common to all 3; only one workstation supported synchronous transfers at the time of the experiments, which would improve the relative performance of that station. However, data transfer is only one part of the more complex SCSI command protocol. The discussion measures performance in megabytes per second (mbs), kilobytes (kb) and milliseconds (ms).

6.1 I/O Transfer Rates

Figure 6 illustrates the data transfer rate of the 3 workstations and ION. The abscissa is indexed in disk sectors of 512 bytes, which corresponds to the physical block size used on most SCSI drives. Most Unix file system traffic occurs in 16 sector increments. This measurement shows only data transfer rate, and does not include any additional SCSI command overhead. It is therefore indicative of the maximum performance attainable by each system. The expected profile of these plots would be rapid increase in transfer rate up to the maximum data rate, followed by steady performance. Instead it can be seen that the rise is fairly slow, indicating additional overhead in setting up the DMA phases of the transfers. Also, the *read* plots indicate crossings in the data rates at different transfer values, rather than a constant ordering from the slowest to the fastest system.

6.2 Command Overhead

Figure 7 illustrates the command overhead component of a data transfer. These data are obtained by subtracting the data transfer time from the command time. System C, ION and System A indicate the ex-

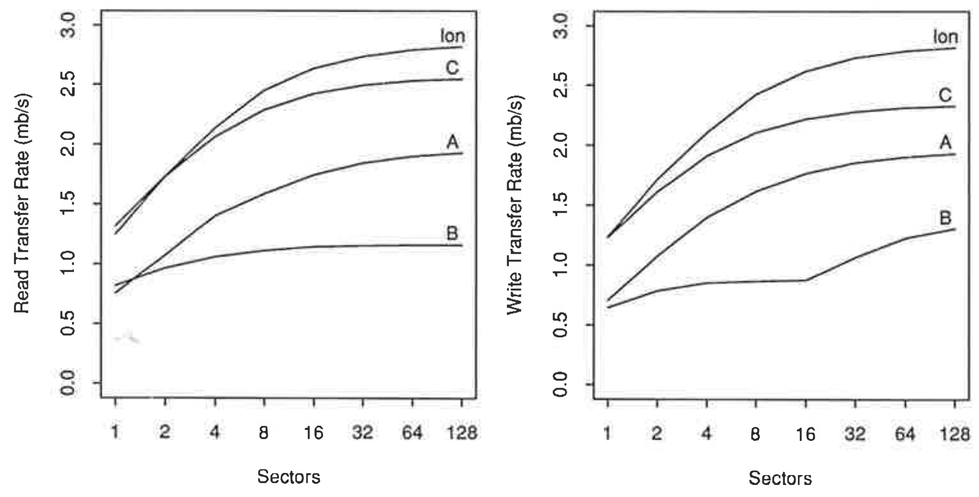


Figure 6. I/O transfer rates. These plots do not include other SCSI phase overhead.

pected overhead profile—an almost constant component of the overall command. System B exhibits peculiar behavior between 2 and 16 sectors, where the command overhead increases sharply and then flattens out. We speculate that System B does not contain DMA control in the SCSI data paths, but includes a FIFO (a hardware device used as a temporary buffer for fast I/O data). When the FIFO reaches its capac-

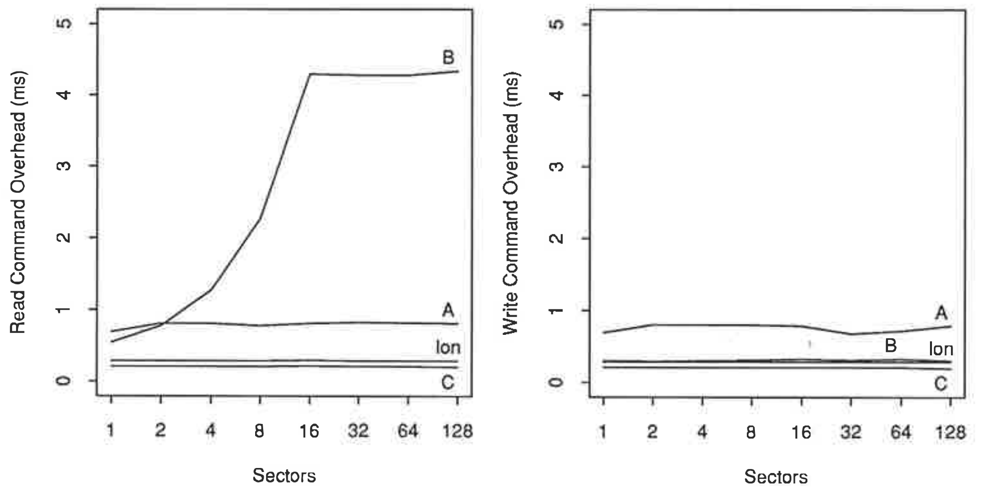


Figure 7. Command overhead. These plots illustrate command time less data transfer time.

ity, it must be drained by a programmed copy loop executed by the processor, thereby increasing the overhead associated with larger transfers.

6.3 State Machines v. Multiple Tasks

While the ION SCSI interface is the fastest of the systems shown, in Figure 7, System C is consistently faster. The explanation involves the different methods in which the SCSI protocol is implemented in each system. Most of the additional overhead for a transfer is caused by the phase changes (and their accompanying interrupts) in the SCSI protocol. These changes can be handled more rapidly with a state machine implementation (typical of most Unix systems) than by the multiple tasks in use in the ION system. In a state machine, the phase transitions can be controlled at interrupt time, while ION must incur the overhead of task synchronization and task scheduling before the transition can occur.

7. A Memory-Based Pseudo-Disk Application

Another ION application is to function as a pseudo-disk for analyzing computer system behavior as disk technology changes. By defining hardware disk characteristics in software, it is possible to use ION to study the impact of new disk technology before it is commercially available. When such an application is backed by sufficient buffer memory in ION, actual file-system behavior, rather than simulation, can be monitored.

Using software running within ION, the following attributes can be controlled:

- Rotational latency
- Head positioning time
- Transfer rate
- Sectors per track
- Tracks per cylinder

In addition, the behavior of various caching strategies within a disk drive can be studied for their affect on device throughput.

7.1 A Zero Latency Disk

If the above parameters are adjusted for maximum performance, a zero-latency pseudo-disk drive results. It is similar in effect to a memory based file system [22] in that physical I/O operations are replaced by access to RAM memory. Three experiments were run to determine the impact of such a device in a Unix environment: building the ION source, copying a 2 megabyte file, and database-like accesses involving widely scattered I/O operations. The latter consisted of 4000 single sector reads uniformly distributed over a disk partition of 32768 sectors. All tests were conducted on an idle workstation. The pseudo-disk experiments differed from the real disk case by moving all the executable images, source files and temporary file storage onto the pseudo-disk.

Operation	Real	User	System	Improvement
Build ION—Real Disk	156.7	81.2	28.2	
Build ION—Pseudo-Disk	126.6	81.0	27.7	1.2
Copy File—Real Disk	11.5	0.0	1.6	
Copy File—Pseudo-Disk	7.2	0.0	1.6	1.6
Random Access—Real Disk	67.7	0.1	4.2	
Random Access—Pseudo-Disk	8.1	0.1	2.7	8.4

Table 2. Real disk v. pseudo-disk performance. Time measured in seconds.

The results indicate that this type of pseudo-disk is not very practical for single-user operations involving sequential file access. The readahead/write-behind strategies employed in Unix work exceptionally well when the CPU performs some processing of the data between I/O accesses. A better improvement is seen when only minimal processing occurs, as in the file copy. For random behavior, the pseudo-disk is seen to behave significantly better than a conventional disk drive. Further study is necessary to compare such results when multiple users are involved, for example, on a file system server. When requests for sequential file access occur from multiple sources, it is possible that the resultant behavior more closely resembles scattered accesses as requests are intermixed.

8. Availability

ION is available for experimental research use. Interested parties should contact the author.

9. Conclusions and Future Work

ION is far from being a completed project. The system is evolving as it accommodates additional peripheral devices and application functions. ION has proved to be a flexible tool for experimenting with new hardware, especially given its nonintrusive (to the host workstation) development environment.

There are several additional features that are necessary to improve the utility of ION. Dynamic task definition, implemented either through an interpretor or by using down-loaded compiled code, would simplify the development process. A graphical interface, instead of the in-line description language, would also be a more intuitive task and queue description mechanism and would be less error prone. Work is underway to construct an object repository for storing various size intermediate and long-lived data objects. It will be accessible from both internal and external ION applications.

10. Appendix—What is SCSI?

SCSI, the Small Computer System Interface, is a protocol definition for connecting processors, disk drives, printers and other devices. It is a high-level interface that expects a significant amount of intelligence within the controller associated with each device. This is in sharp contrast to other disk interfaces (e.g., SMD) where individual devices typically respond only to control signals, and all programmed intelligence resides in the host controller. Up to 8 devices can exist on a single SCSI bus, each taking a fixed SCSI device identifier number. Most hosts insist on being device 7. Each device can be composed of up to 8 independent subdevices using a logical unit number (LUN) facility. However, few devices and operating system implementations support this feature. The maximum bus length is about 20 feet, although a differential bus specification also exists which permits a total bus length of 80 feet.

10.1 SCSI Devices, Commands and Phases

SCSI devices are typically classified as either initiators or targets, although these roles need not be permanent. As the names imply, an initiator (usually the host processor) starts an operation by arbitrating for the SCSI bus and selecting a target device (such as a disk drive) to respond to its request. All further action is controlled by the target device which indicates its intentions by changing the SCSI bus phase.

A facility known as *disconnect/reconnect* allows better utilization of the SCSI bus. If a command involves a relatively long delay before requested data will be available, the target can disconnect from the SCSI bus, making it available for other targets, and reconnect when the data are ready. Such delays are normally encountered during physical head repositioning on disk drives. The initiator informs the target of its ability to accommodate this behavior during a message exchange before the actual command begins.

Six phases are defined by the SCSI specification to coordinate transmissions between the initiator and the target. The terminology of *in* and *out* used below is always with respect to the initiator. All phase changes are controlled by the target.

command	The target is requesting a multibyte command sequence that defines the desired operation.
status	The target is returning a single status byte to the initiator indicating the outcome of the command.
message in	The target is sending a control message to the initiator. Messages are transmitted to indicate parity error detection, command completion and identification of sub-units within a target.
message out	The target is requesting a message from the initiator. This phase is usually generated in response to a control signal (*attention*) asserted by the initiator.
data in	The target is instructing the initiator to begin accepting data as a result of the command.
data out	The target is requesting data from the initiator, as described by the command.

10.2 Whither SCSI?

The next generation of SCSI, SCSI-2, is a mostly upwards compatible change with many optional SCSI-1 commands and messages becoming mandatory. The significant improvements involve the width of the data path and the cycle time for each individual transfer on the bus. SCSI-1 has an 8 bit data path with a minimum cycle time of 200 nanoseconds yielding a maximum throughput of 5 mbs. SCSI-2 can use optional secondary cables, providing 16 or 32 bit transfers. In addition, the minimum transfer cycle time is reduced to 100 nanoseconds. Hence, the maximum throughput is 40 mbs. SCSI-2 peripherals and controllers are beginning to appear on the marketplace.

SCSI-2 also provides a mechanism for command queueing, where an initiator can send multiple commands to a target, allowing it to service these requests in a device specific optimal ordering. A further feature allows a target to inform an initiator of a change of condition, even if the initiator does not have a command pending with the device. This is instrumental in returning error conditions such as device off-line which formerly required polling of the target.

References

[1] ANSI X3.131, Small computer systems interface (SCSI), American National Standards Institute, Inc.

[2] S. Leffler, M. McKusick, M. Karels and J. S. Quarterman, *Design and Implementation of the 4.3 BSD Operating System,* Chapter 9, pp 225-257, Addison Wesley, 1989.

[3] D. D. Clark, The structuring of systems using upcalls, *Proc. 10th Symp. on Operating Systems Principles,* Washington, December 1985.

[4] A. Birrel and B. Nelson, Implementing remote procedure calls, *ACM Trans. on Computer Systems,* 2(1):39-59, Feb. 1984.

[5] J. Northcutt, Mechanisms for reliable distributed real-time operating systems—the Alpha kernel, Academic Press, 1987.

[6] H. Tokuda and C. Mercer, ARTS: A distributed real-time kernel, *Operating Systems Review,* 23(3):29-53, July 1989.

[7] P. Gopinath and K. Schwan, Chaos: Why one cannot have only an operating system for real-time applications, *Operating Systems Review,* 23(3):106-125, July 1989.

[8] M. Accetta et. al., Mach: a new kernel foundation for Unix development, *Proc. Usenix Summer Conference,* Atlanta, GA, 1986.

[9] C. Wilkenloh et al., The clouds experience: building an object-based distributed operating system, *Proc. Distributed and Multiprocessor Systems Workshop,* 333-347, Fort Lauderdale FL, October 1989.

[10] J. Stankovic and K. Ramamritham, The design of the Spring kernel, *Proc. Real Time Systems Symp.* 146-155, CA, December 1987.

[11] C. Pu, H. Masselin and J. Ioannidis, The Synthesis Kernel, *Computing Systems,* 1(1):11-32, Winter 1988.

[12] D. Cheriton, The V kernel: a software base for distributed systems, *IEEE Software* April 1984.

[13] J. Stankovic and K. Ramamritham, The Spring kernel: a new paradigm for real-time operating systems, *Operating Systems Review,* 23(3):54-71, July 1989.

[14] R. Babb II, Parallel processing with largegrain data flow techniques, *IEEE Computer,* 17(7), July 1984.

[15] R. Rasmussen et. al., Max: Advanced general purpose real time multicomputer for space applications, *Proc. Real Time Systems Symposium,* San Jose, CA, December 1987.

[6] D. Ritchie, A stream input-output system, *AT&T Bell Laboratories Technical Journal,* 63(8), October 1984.

[17] Sun Microsystems, Inc., Network extensible file system protocol specification, Draft, 1990.

[18] Desklab™ Real-time Analog Data I/O, Gradient Technology, Inc. Burlington, NJ.

[19] A. Park, J. Becker and R. Lipton, IOStone: A synthetic file system benchmark, *Computer Architecture News,* 18(2):45-52, June 1990.

[20] B. Wolman and T. Olson, IOBench: A system independent IO benchmark, *Computer Architecture News,* 17(5):55-70, 1989.

[21] J. Ousterhout et. al., A trace driven analysis of the Unix 4.2 BSD file system, *Proc. 10th Symp on Operating System Principles,* December 1985, Washington.

[22] D. Ritchie and K. Thompson, The UNIX timesharing system, in *Bell System Technical Journal,* vol. 57, no. 6, part 2, July-August 1978.