Robert J. Walsh

DART: Fast Application-level Networking via Data-copy Avoidance

The goal of DART is to effectively deliver high-bandwidth performance to the application, without a change to the operating system call semantics. The DART project was started soon after the first DART switch was completed, and also soon after line-rate communication over DART was achieved. In looking forward to gigabit class networks as the next hurdle to conquer, we foresaw a need for an integrated hardware-software project that addressed fundamental memory bandwidth bottleneck issues through a system-level perspective.

The Ethernet supported large 100-node networks in 1976.[1] By 1985, 10 Mb/s Ethernet had been available for a while, even for PCs. However, high-performance hardware and software lagged, due to system bottlenecks above the physical layer. The premier implementations for UNIX were achieving only 800 kb/s (8 % of 10 Mb/s) in benchmark scenarios on common system platforms of the day.[2]

The deployment of 100 Mb/s fiber distributed data interface (FDDI) provided an order of magnitude bandwidth increase in the link speed around 1987. However, the end system could not saturate the link on generally available machines and operating systems until 1993,[3] when Transmission Control Protocol (TCP) improvements and a CPU capable of 400 million operations per second became available.[a] Once again, high-performance hardware and software lagged the potential provided by the physical layer.

The current technological approach is switching. Gigabit-class links and adapters, such as 622 Mb/s asynchronous transfer mode (ATM), are becoming available. Since ATM links are dedicated point-to-point connections, the use of 622 Mb/s in switch-to-switch links and at the periphery implies that one ought to be able to move data at gigabit rates.

Switched capacity promises a lot to servers; however, mainstream systems are not currently capable of effectively using the bandwidth. The DART project attempts to avoid the Ethernet and FDDI scenarios where end-system performance lags physical-layer potential.

One of the early goals was to go beyond simple benchmark scenarios where line rate communication connects a phony bit source to a phony bit sink, with the CPU saturated. The context for the work was to connect two applications at high speed, leaving CPU

[a]The TCP improvements included a small architectural update, the window scaling extension, to abstractly support the advertisement of more than 64 kbytes of receive buffering. The rest of the improvements derived from implementation efforts to increase the actual buffering allocated to advertised TCP windows, and to improve the segmentation of the TCP byte stream into packets.

resources available to execute the applications. In the past, the CPU had been saturated in Ethernet and FDDI quests for line rate communication.

Layering

The motivation for DART arises from the specific layering and abstraction used in BSD-derived UNIX systems, but the context is sufficiently general that the problem and solution have wide applicability. Since various layers within system software will be referenced repeatedly, we introduce them using Figure 1.

The *application* generates and consumes data. It tells the operating system which data to communicate when, by using read and write system calls.

The *socket layer* moves data between the operating system and the application. It also synchronizes the application with the networking protocols based on data and buffer availability.

The *transport protocol layer* provides a connection to the remote peer. In the case of TCP, the connection is a reliable byte stream. TCP takes on the responsibility of retransmitting lost or corrupted data, and of ignoring reception of retransmitted data that was previously received.

The *network protocol layer* provides an abstract address and path to the remote host. It hides the various hardware-specific addresses used by the various media in existence. In the case of IP, fragmentation allows messages to traverse media which have different frame sizes.

A conventional *driver layer* moves data between the network and the system. It uses buffers and data structures whose representation percolates throughout all the operating system networking layers.

The DART Concept

DART increases network throughput and decreases system overheads, while preserving current system call semantics. The core approach is data copy avoidance, to better utilize memory bandwidth.

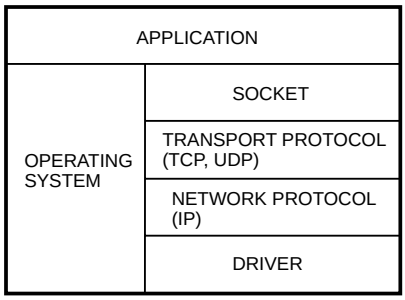

Figure 1
Software Layering

Memory bandwidth is a scarce resource that must not be squandered. In DIGITAL's transition from MIPS processor systems to Alpha processor systems, CPU performance increased more rapidly than main memory bandwidth. It took approximately 340 μs to move 4500 bytes on the MIPS-based DECstation 5000/200, and approximately 200 μs on the Alpha-based DEC 3000/500. In the same time, the fixed per-packet costs were reduced by a factor of three or more. (General trends are also stated in Reference 4.)

One breakdown of networking costs is reported in Reference 5. The variable per-byte costs reported there are all associated with memory bandwidth, which is improving slowly. The fixed per-packet costs in the driver, protocol, and operating system overhead are all generally associated with the CPU, which is improving rapidly. Thus, we focus on the per-byte memory bandwidth issues as those most needing architectural improvement.

A traditional system follows the networking subsystem model implemented within the BSD releases of UNIX, shown in Figure 2. An application uses the CPU to create data (1), the socket portion of the system call interface copies the data into operating system buffers (2 and 3), the network transport protocol checksums the data for error detection purposes (4), and the device driver uses programmed input/output (I/O) or direct memory access (DMA) to move the data to the network (5). Graphs showing the dominant costs of checksumming and kernel buffer copies are presented in Reference 6.

These five memory operations are a profligate waste of memory bandwidth. A system with a 300 Mbyte/s memory system would achieve at most 300*8/5 = 480 Mb/s I/O rates. The system would be saturated.

The DART model is shown in Figure 3. The DART model is that data is created (1) and sent (2). Two memory operations make efficient use of the memory bandwidth.

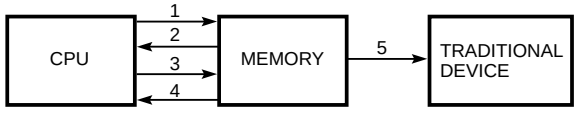

Figure 2
BSD Copy-based Architecture

Figure 3
DART Zero-copy Architecture

Squandering of memory bandwidth is avoided. A system with a 300-Mbyte/s memory system would encounter the larger bound of 300*8/2 = 1200 Mb/s for I/O rates. Resources are available for the application even when running at line rate.[b]

To support the DART concept, we need a system perspective that integrates the hardware and software changes implied by the DART model. Hardware is responsible for checksumming instead of software. Hardware is solely responsible for data movement, instead of redundant actions by both hardware and software. These hardware changes are bounded and generic.

Operating system software retains the application interface and general coding of the BSD UNIX implementation. Extensive changes are unnecessary, since the focus is the core lines that represent data movement consumption of memory bandwidth. Extensive changes are also undesirable, since there is a large base of software written to the current properties of the BSD networking subsystem.

The DART Hardware

The first implementation of the DART concept is a high-performance 622-Mb/s ATM network adapter for the PCI bus called DART. DART's design reflects an awareness of the interactions of the components of the system in which it is placed. The PCI bus, main memory, cache, and system software can all be used efficiently.

Store-and-Forward Buffering and DMA

DART is an adapter that connects a gigabit-class network to a gigabit-class I/O bus, and is appropriate for systems with gigabit-class memory systems. DART is focused on the server market where a slight increase in adapter cost can be acceptable if the system performance is significantly improved, since main memory and other costs dominate the cost of the DART adapter.

DART alleviates main memory bottlenecks through a store-and-forward design, as shown in Figure 4. Traditional networking software subsystems and applications perform at least five memory operations to create, copy, checksum, and communicate data. DART's *exposed buffering* allows data to be created and communicated with just two main memory operations. The adapter memory is a resource that can be better utilized by exposing it to the operating system, and better performance results as well. This is similar to the exposure of the CPU-internal mechanism in the CISC-RISC (complex to reduced instruction set) transition.

DART contains a number of features to make the store-and-forward design effective. DART's bus master and receiver summarize network transport protocol checksums for software. DART's bus master provides byte-level scatter-gather data movement to support communication out of application buffers, not just operating system buffers. DART provides packet headers for software to parse so that software can direct the bus master to place received data in the application's buffers when the application desires, without operating system copy overhead.

Buffering Design An ATM segmentation and reassembly (SAR) chip accesses virtual circuit state for each cell, and operates on 48-byte cell payloads. The payload naturally corresponds to a burst-mode operation, leading to the use of synchronous dynamic DRAM (SDRAM) to buffer cells. The circuit state is generally smaller and randomly accessed, leading to the use of static RAM (SRAM) for control information. Dividing the data storage architecture into two parts allows the interface designs to be tailored to the characteristics of the data type in question.

The DART prototype uses 16 Mbytes of SDRAM for the data memory. The prototype uses 1 Mbyte of SRAM for the control memory. The SDRAM supports hardware-generated transmissions, aggregation of data for efficient PCI and host memory interactions;[c] and buffering for received data until the application indicates the proper destination for it. The SRAM contains the SAR intermediate state; with a large number of virtual circuits and ATM's interleaving of packet contents, there is too much state to be recorded on-chip at this time.

Packet Summarization for Software The receiver parses the cells for the various packets which are interleaved on the network connection, and reassembles the cells into packets. Once all the cells composing a packet have been received, a packet descriptor is prepended to the packet. The descriptor contains length, circuit number, checksum, and all other information that the driver may need to parse and process the packet.

Upon packet reassembly, a hardware-initiated DMA operation moves software-configured amounts of descriptor and packet contents to host memory. When

[b]The 1200-Mb/s figure includes the cost of having the application write the data to memory. Some memory bandwidth might be consumed to fill the CPU's cache in order to execute the application and operating system. In this scenario, if non-network bandwidth is greater than 300*8 — 2*1000 = 400 Mb/s, data production would be the bottleneck and the network would run at less than line rate. This is beneficial; the bottleneck has been moved to the application.

[c]Some adapters segment (or reassemble) from host memory, leading to 48-byte payload transactions with host memory. Transaction size should be an integral multiple of the cache block size, and should be aligned, in order to avoid wasting system bandwidth.

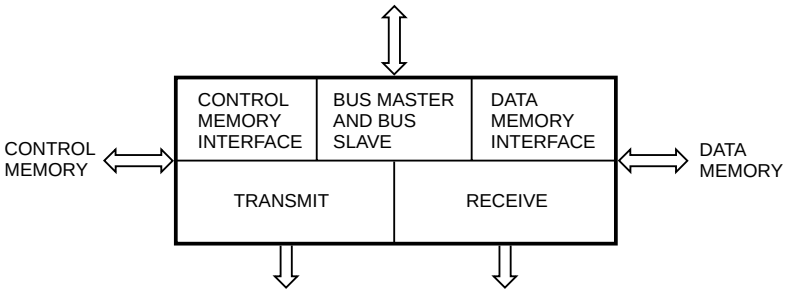

Figure 4
DART Block Diagram

properly configured, the hardware provides the network and transport headers, allowing software to determine where to place the packet data. Software data copies are avoided by allowing software to initiate a DMA operation to move the data to its final application-desired location, rather than to some expedient, but inefficient, operating system buffer.

Receive Buffering DART's store-and-forward receive buffers are divided into two classes. The per-circuit class guarantees each circuit forward progress. Each circuit is individually allocated some buffers in which to store cells. No other circuit can prevent data from passing through such buffers. The shared class is preferentially used, and avoids resource fragmentation problems. Any circuit can consume a shared buffer for an incoming cell.

Since software specifies when and where to store packet data, adapter buffers are recycled when software decides to do so, and not independently by hardware. Part of a packet may be stored in application buffers at one time, and other parts of the same packet may be stored in application buffers at later times. Hardware cannot assume a one-to-one correspondence between receive DMA and complete packet consumption.

Flow control occurs in the socket layer based on transmit buffer availability, in the transport layer based on remote receive buffer availability, in the driver based on adapter resource availability, and in the ATM layer based on cell buffer availability within the network. Credit-based flow-control protocols for ATM are based on the source of a cell stream on a link decreasing a counter (quota) when a cell is sent, and increasing a counter when a credit is received.[7] The decrement represents buffer consumption at the next hop. The credit advertises buffer availability to the source; the next hop has forwarded a cell and thus freed a buffer.[d]

[d]Forwarding the cell is required for (per-circuit) buffers of which the transmitter on the link was made aware during link initialization. The receiver on the link can generate credits immediately for (shared) buffers hidden from the transmitter during link initialization.

With FLOWmaster, the credit is conveyed across the link to the source of the cell stream by overlaying the virtual path identifier (VPI) field with the circuit to credit. This is a nonstandard optional use of the ATM cell header. Quantum Flow Control is a credit-based flow-control protocol for ATM that batches the credits into cells instead of overlaying the VPI field.

Since credit-based flow-control is based on buffer availability, credits advertising free buffers can potentially be held up by software actions. The shared class allows immediate credit advertisement, and best enables line rate communication. The per-circuit class involves software packet processing in the credit advertisement latency. To advertise a credit for a circuit whose per-circuit quota is exhausted, either the circuit must recycle an adapter-buffered packet, or any circuit must recycle a shared-class, adapter-buffered packet.

A minimal memory that constantly ran out of per-circuit buffers and flow-controlled the source would exhibit poor performance. DART uses a large data memory. Advertising (shared) buffers via credits keeps the data flowing through the overall network and systems with high performance.

Transmit Buffering Software performs all transmit buffer management. Software creates a free buffer list of its own design, allocates buffers from the list to hold packet data, and recycles buffers after observing packet completion events. Software makes the trade-off between large efficient buffers which may be incompletely filled, and small buffers which waste less storage but incur increased allocation, free, DMA specification, and transmit description overheads.

Peer-to-Peer I/O

DART avoids system resource consumption in server designs by supporting peer-to-peer I/O. A traditional server would consume PCI bus and main memory bandwidth twice by using main memory as the store-and-forward resource between two I/O devices, as shown in Figure 5. The PCI bus is consumed during steps 2 and 5. The main memory is consumed during

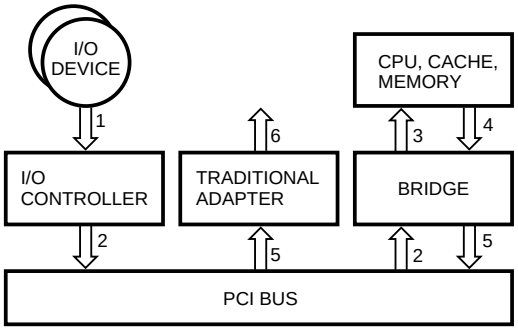

Figure 5
Traditional Server Architecture

steps 3 and 4. On some systems, I/O operations compete for cache cycles during steps 3 and 4, whether the cache is external to or internal to the CPU. Such resource consumption can cause the CPU to stall even though the CPU will never examine such data.

DART allows a single PCI bus transaction to move the data, as shown in Figure 6. This also avoids any main memory bandwidth consumption when a bridge isolates the PCI I/O bus from the main system bus. The cache is not consumed with nuisance coherence loads for data the CPU will never examine, and the CPU does not have to contend with I/O for cache or main memory cycles.

For peer-to-peer I/O over DART, the CPU is only involved in initiating packet transmission. This is a relatively small burden, since only a little bit of control information needs to be computed and communicated to the adapter.

To enable efficient peer-to-peer I/O, DART includes a bus slave as well as a bus master. *The bus slave makes the internal resources of the adapter visible on the PCI bus* through DART's PCI configuration space base address registers. Therefore, on the PCI bus, the data memory looks like a linear contiguous region of memory, just as main memory does. The bus slave supports both read and write operations for these typically internal resources.

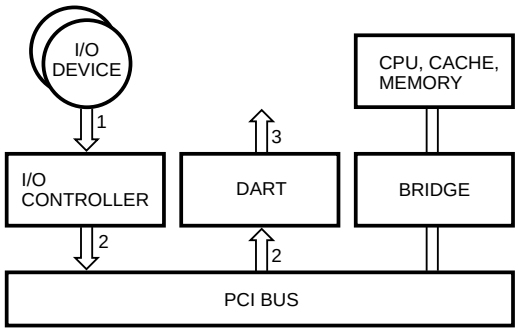

Figure 6
DART Server Architecture

DART provides efficient handling of small packets. Typically, describing a number of small packets for transmission is onerous for software, limiting the peak packet rate. DART's transmitter can automatically subdivide a large amount of data into small packets, eliminating a lot of per-packet overhead. This feature is appropriate for a video server, whose software cannot possibly fill the network pipe if it must operate on 8-cell packets.

PCI Interface

DART supports both 64- and 32-bit variants of the PCI bus. The network interface and DART memories provide prodigious bandwidth. To fully take advantage of them, a 64-bit PCI bus is recommended, but DART will also operate on a 32-bit PCI bus.

Bus Reads and Writes The DART architecture supports memory write-and-invalidate hints to the bridge between the system bus and the PCI I/O bus. Such a hint informs the bridge that the I/O device is only writing complete cache blocks. There is no need for read-modify-write operations on main memory cache blocks in such circumstances.

Write operations within a system are generally buffered. A path from the origin of the write to the final destination can be viewed as a sequence of segments. As data flows through each segment, each recipient accepts data with the promise of completing the operation, allowing each source to free resources and proceed to new operations. Thus, write paths are generally not performance-limiting as long as there is sufficient buffering to accept burst operations. In the DART context, the bridge between the system bus and the PCI I/O bus accepts DART's writes and provides buffering for high throughput.

However, read operations are more problematic. When memory locations are shared between CPUs, caches may or may not be kept coherent by hardware. Here, the memory locations are shared between the CPU and I/O device, and there is no coherence support. Each DART read suffers a round-trip time through the bridge to access the main memory. DART addresses this latency through large read transactions (up to 512 bytes).

As an example, consider a simplified 64-bit bus where 540 Mb/s of data are written in 64-byte bursts, reads suffer 15 stall cycles until the data starts to stream, and writes require a stall cycle for the target to recognize its address. Address and data are time-multiplexed at 33 MHz. Then writes consume 540 * (1 + 1 + 8)/ 8 = 675 Mb/s of bus bandwidth. Reads have 33 * 8 * 8 — 675 = 1437 Mb/s of bus bandwidth into which they must fit. Thus, the minimum burst length L required is $540 * (1 + 15 + L) = L \leq 1437$. The burst must be at least 9 cycles, 72 bytes, in the ideal case. DART's large read burst size compensates for overheads like large read latencies.

Importance of Bus Slave Interface The bus master interface is appropriate for software-generated transmissions. A packet created by an application in main memory can be moved via DMA to the network.

The bus slave interface is appropriate for hardware-generated transmissions. Another I/O device which is designed to always be bus master, like a disk interface, can move data directly to the DART without intermediate staging in a memory. Peer-to-peer I/O, however, was a by-product of other concerns.

Data transfer within TCP is based on a stream of large data packets flowing in one direction, and a stream of small acknowledgments flowing in the opposite direction. Traffic analysis studies often find a mix of smaller and larger packets. One of the early concerns for the DART project was to make transport protocol generation of acknowledgments inexpensive by avoiding DMA. A small packet, constructed entirely by the CPU anyway, could be moved to the I/O device instead of to main memory. This is fundamentally a short sequence of write operations that could easily be buffered, allowing the CPU to proceed in parallel on other work.

DMA from an application buffer to a device interface is generally specified to hardware by stating the physical addresses of the application buffer in main memory. DMA requires a guarantee that the data is at the specified locations. If the virtual memory system were to migrate the data to disk and recycle the physical memory for some other use, the parallel DMA activity would move the wrong data. Therefore, DMA operations are surrounded by page lock and unlock calls to the virtual memory system, to inform it that certain memory locations should not be migrated.

Additional concerns that led to incorporation of the bus slave interface were related to the cost of page locking, and the cost of acquiring and releasing DMA resources (e.g., in the bridge). An acknowledgment might be constructed in nonpaged kernel memory, but a small application packet would likely be constructed in application memory subject to paging. Even if page locks were cached for temporal locality, it might be cheaper to simply move the data via programmed I/O.

The break-even point between DMA and programmed I/O is system-dependent, but can be measured at boot time in order to learn an appropriate threshold to use for such a decision. Demands on the main memory system from its various clients will change over time, and a single measurement is only optimal for the sample's conditions. The suggestion here is to enable a quick judgment in the software. The intent is to make large gains and avoid egregious performance errors. We suspect that fine-tuning the decision is less important, and requires the collection of excessive information during the normal operation of the system.[c]

Interrupt Strategy As noted above, on-chip access rates for the CPU increase more quickly than off-chip access rates. Interrupt processing and context switching are fundamentally off-chip actions; new register values must be loaded into the CPU, and the cache must be primed with data. Thus, the general system trend is that interrupt processing and context switching improve more slowly than raw processing performance.

DART provides a programmable interrupt holdoff mechanism. By delaying interrupts, events can be batched to reduce various system overheads. If the batching mechanism were not present, an interrupt per packet would swamp system software at gigabit rates.

Since the interrupt delay interval is programmable, software may use adaptive algorithms to decrease interrupt latency if the system is idle, or to increase the amount of batching if the system is busy. The delay timer starts decrementing as soon as it is written. Typically, the timer will be written at the end of the interrupt service routine.

Interrupts can be divided into two classes by software. Each class has its own delay interval, in case software assigns distinct importance or latency requirements to the classes.

The Dart Software

DART provides increased performance with the same system calls, and with the existing system call semantics. The only change is to the underlying implementation of the existing system call semantics.

Unmodified existing applications can consume gigabit network bandwidth. The application can assist the system software by using large contiguous data buffers, but it is not required. System software can specify byte-level scatter/gather operations to the DART adapter in order to access arbitrary application buffers.

Changes to the system software are confined to a few locations above the driver layer, and are generic. Successive high-bandwidth adapters for other media can be supported by just writing drivers; no changes will be needed above the driver layer. The shared set of upper-layer software changes are only needed to take maximum advantage of a DART-style adapter; a traditional copy-based implementation is supported by the hardware.

[c]Given the parallel nature of the environment (other I/O, cache operations, and multiprocessor CPUs), a software system could only estimate non-DART memory loads. Queued DMA operations may start later than expected, or finish before their completion has been noticed. CPU cache activity is dependent on the program executing at that moment; fine-tuning is problematic. The focus of DART has been the large gains, like avoiding copies, or allowing either DMA or programmed I/O to be used. The focus has been on the structure of the system.

We developed a prototype UNIX driver to test the upper-layer changes, and executed a modified kernel against a user-level behavioral model of a DART-style adapter. The code was subjected to constant background testing on a workstation relied on for daily use. The prototype driver supports buffer descriptors referencing either kernel buffers or adapter buffers. The implementation effort to support kernel-buffered packets was minimal, and enables multiple protocol families to be layered above the driver.

The software changes modify the existing upper-level software, rather than bypassing it via a collapsed socket, transport, network, and driver implementation. The current UNIX networking subsystem provides a rich set of features that needs to be completely supported for backward compatibility.

Transmit Overview

A comparison of traditional transmission with DART transmission is shown in Table 1. For a traditional adapter, the system call layer copies application data to operating system buffers. With a DART adapter, the data is copied to the adapter. Uiomove is the copy function typically used within UNIX. The DART mechanism is to use an indirect function call through a pointer, rather than a direct function call to an address specified by the compiler's linker. High-performance copy functions are associated with the device driver. The driver's copy function is free to use DMA or programmed I/O, depending on the length of the copy.

For a traditional adapter, software wastes machine resources computing checksums. With a DART adapter, the checksum is computed by hardware as the data flows into the adapter. The adapter can patch the checksum into the packet header. The adapter can also move checksum summaries back to host memory so that they are available for retransmission algorithms.

For a traditional adapter, the driver instigates additional memory references to copy the data to the adapter for transmission. With a DART adapter, the data is already on the adapter, ready to be sent! Much of the data copy avoidance work is throughput-related. In this instance, we also create the potential for a latency advantage for the DART model, since the data copy overlapped work in the system call, transport, network, and driver layers of the operating system.

Receive Overview

In many ways, the receive path for networking is usually considered more complicated than the transmit path, since the various demultiplexing and lookup steps are based on fields that historically have been considered too large to use simple table indexing operations. Also, the receive path requires a rendezvous between the transport protocol and the application (to unblock the application process upon data arrival). So it should come as a pleasant surprise that the DART-style changes for packet reception can be as simple and localized as two conditionals in the socket layer and one in the network transport layer.

Table 2 is a comparison of traditional receive processing with DART receive processing. It is almost identical to the packet transmission comparison. The distinction is which portion of the DART adapter computes the checksum on behalf of the software (receiver instead of DMA engine).

Interrupts

Transmit completion interrupts do not need to be eagerly processed. Software can piggyback processing to reclaim transmit buffers upon depletion of transmit buffer resources, upon unrelated packet reception events (e.g., User Datagram Protocol, UDP), and upon related packet reception events (e.g., TCP acknowledgment). The transmit completion events can be masked, or the hardware interrupt holdoff mechanism can be used to give them a longer latency.

Receive interrupts are batched to reduce overheads. Short packets are fully contained in the initial packet summary which would be deposited in a kernel buffer. Adapter buffers for short packets can be recycled immediately by system software. Long packets are not fully contained in the initial packet summary provided software for parsing and dispatch. The summary is noticed during one interrupt, and scatter/gather I/O completion into application buffers is noticed during another interrupt if performed asynchronously.

The side-effect of the decision to create a store-and-forward adapter is that a received packet is related to two interrupts. The intent is *not* to burden a system and cause multiple interrupts per packet. The distinction between *relation* and *causality* is important.

When the system is under load, there is a steady stream of packets, and thus a steady stream of batched

Table 1
Transmit Overview

	Traditional	DART
System call layer	Uiomove user buffer to kernel buffer	*Uiomove user buffer to adapter buffer
Protocol layer	For all buffers for all bytes, update checksum	For all buffers, update checksum
Driver layer	Programmed I/O or DMA	Data is already on the adapter!

Table 2
Receive Overview

	Traditional	DART
Driver layer	Programmed I/O or DMA	Data stays on adapter!
Protocol layer	For all buffers for all bytes, update checksum	Use checksum computed by receiver hardware as packet was reassembled
System call layer	Uiomove kernel buffer to user buffer	Uiomove adapter buffer to user buffer

interrupts. If 3 Mbytes were transferred using a burst of 1-kbyte packets, there would be 3000 packets. Batching 20 packets/interrupt, there would be 150 interrupts to report packet arrivals. The first interrupt is just for packet arrival events, to allow header parsing. The intent is for the next 149 interrupts to report 20 new arrivals and the DMA completion for 20 previous arrivals. A final interrupt would take care of the final DMA requests. In this case, the additional interrupt load for a DART adapter is minor: one interrupt for 3000 packets. The interrupt load is not doubled (even if one chooses to move received data asynchronously).

Store-and-forward latency is incurred because of the memory write and read on the adapter (to store data from the network and to later move it to the application's buffers). DART adapter memory operates at a high rate, over 4 Gb/s, to minimize this. Due to the intervening software decision concerning where to place DART data for large packets, the data may be placed at its initial location in host memory later than for a traditional adapter which fills kernel buffers. However, store-and-forward reduces main memory bandwidth consumption, and quickly places the data at its *final* location within the application buffers in host memory. The correct metric is latency to data availability to the application, not data latency to first reaching the system bus.

CSR Operations

Control and status registers (CSRs) are used within hardware implementations to allow software to control the action of hardware, and for hardware to present information to software. For example, a CSR can inform a device of the device's address on a bus. In this case, the CSR's definition is generic in the context of the bus definition. Alternatively, a CSR can be used to initialize a state machine within the hardware implementation. In that case, the CSR's definition is specific to that version of the device.

CSR reads are very expensive. Generally, a single CSR read is required for DART interrupt processing, and that CSR is placed in the PCI clock domain of DART in order to avoid operation retries on the PCI bus.

Most packet processing information is written to host memory by the adapter for quick and easy CPU access. For example, packet summaries are placed in one or more arrays in host memory, and software can use an ownership bit in each array element to terminate processing of such an array.

CSR writes are buffered; nevertheless, they can be minimized. The packet summaries in host memory are managed with a single-producer, single-consumer model. When the consumer and producer indices into an array are equal, the array is empty. When hardware's producer index is greater, there are entries to be processed by software. (Redundant information in array element ownership bits means that software does not actually need to read the DART adapter to perform the producer-consumer comparison.) When the hardware's producer index reaches the software's consumer index minus one, the array is fully utilized. When software has processed a number of packet summaries, the hardware can be informed that they can be recycled by a single write of the consumer index to the adapter.

The DMA engine processes a list of "copy this from here to there" commands. By supporting a list of operations instead of a single operation, software can quickly queue an operation and move along to its next action without a lot of overhead. The copy commands reside in an array within host memory, with a software-specified base and a software-specified length.

DMA commands also follow the producer-consumer model. However, since instructions are only read by DART, there are no ownership-bit optimizations. To compensate for this, software can allocate a large array and cache a pessimistic value for the hardware's consumer index in order to avoid CSR reads. Alternatively, the DMA engine could periodically be given instructions to DMA such information to host memory.

A typical DART interrupt involves one CSR read and three CSR writes, yielding an efficient interface. One read determines interrupt cause. One write informs the DMA engine of new copy commands for newly received data. Another write informs the DMA engine that the CPU processed a number of the packet summaries DART placed in main memory. A third write initializes the interrupt delay register to batch future events.

Occasionally, an interrupt also involves an extra CSR read. The read discovers a large number of commands processed by the DMA engine, allowing software to recycle entries in the command queue and thereby issue more commands.

Driver

The driver classifies received packets, and decides whether to continue to use adapter buffers for them, or to copy the data into kernel buffers. For the prototype, adapter-buffered packets are:

- Long enough to contain maximal-length IP and transport protocol headers.
- Version 4 IP packets (buffering assumptions percolate throughout the layers of the system, so a protocol family must be updated and tested to support adapter-buffered packets).
- TCP or UDP protocol packets. Other protocols layered over IP do not use adapter buffers, to make the scope of the effort manageable by handling just the common case.

The operating system uses a single *mbuf* to describe a single set of contiguous bytes in a buffer which may be within or external to the mbuf structure. Mbufs can be placed in lists to form packets from a number of noncontiguous buffers.

Received adapter-buffered packets are two mbufs long. The first mbuf contains the initial contents of the packet DMAed into memory by the adapter, that is the protocol headers and summary information from the adapter.

The second mbuf refers to the packet in adapter memory. For ATM, the received packet is stored in a linked list of buffers on the adapter. Programmed I/O access to the buffers requires software to traverse the links, but this would not be done in practice since the CPU read path to the I/O device is unbuffered and high-latency. The DART DMA hardware would be used, and it would traverse the links as-needed. The DMA hardware allows the software to pretend the packet is contiguous.

Fields of the second mbuf are used in specific ways. The length of the second mbuf does not contain the initial portion of the packet copied into the first mbuf, even though the adapter memory buffers the entire packet. The initial portion is replicated, but only the copy local to the CPU is accessed. The pointers of the second mbuf point to bogus virtual addresses, even though the adapter looks like an extension of main memory. This speeds software debugging by trapping inefficient accesses to the adapter. Adjusting the length and pointer fields is still allowed in order to drop data from the front or back of the mbuf. The m_ext fields record the location and amount of adapter buffering used to hold the packet. They also point to a driver-specific buffer reclamation routine.

For TCP, or for UDP packets with nonzero checksums, the driver makes incremental modifications to the DART receive hardware's checksum. The hardware computes the 1's complement checksum over all the cell payloads except for the final ATM trailer bytes. As a result, the driver modifies the hardware checksum to account for:

- Contributions made by IP options
- Construction of the pseudo-header which is not transmitted on the network
- The transport layer checksum, which was zero when the checksum was computed but may be nonzero on the network

To transmit a packet, the transport and network layers operate on protocol headers in main memory. The driver moves the headers to the adapter as part of transmitting a packet whose encapsulated data is in adapter buffers.

The *ifnet* structure is the interface between the protocol layers and the driver. It contains, for example, fields expressing the maximum packet size on the directly connected network, the network-layer address of the interface, and function pointers used to enter the driver.

We add an *(*if_ uiomove)()* field to be associated with buffers as described below. It represents a driver entry to copy data to or from the adapter. We also add an *(*if_ xmtbufalloc)()* field to be used within the mbuf allocation loop of the transmit portion of the socket layer. This allows the socket layer to give precedence to allocating (large) adapter buffers over main memory buffers.

The driver always retains some transmit adapter buffers for its own use. When the system is busy, there will be TCP packets consuming adapter buffers. The packets are associated with the socket send queue. There will also be packets on the interface send queue, which may or may not use adapter buffers. If the first item on the interface queue uses just kernel buffers, then the driver must have reserved adapter buffers in order to complete the transmission and avoid transmit deadlock. At least one packet of adapter buffering must be reserved for the driver output routine.

UDP

UDP motivates many of the changes without getting involved in the complexity of retransmission and reliability. Many of these changes are generic to UDP and TCP: augmenting the buffer and interface descriptions, discovering the availability of efficient buffers for a connection, and allocating and filling the efficient buffers.

One portion of the mbuf is the *struct pkthdr,* which is used only in the first mbuf of a packet. It summarizes interesting information about the packet, like its total length.

We add a *protocolSum* field to the pkthdr of the mbuf so that the driver can communicate the received transport-layer checksum to the upper layers. The transport-layer checksum is not ignored, as it would

be if checksums were negotiated away or cavalierly disregarded. The checksum is verified by the transport layer as usual, but without accessing all the bytes of the packet. The protocolSum field is valid if an *M_PROTOCOL_SUM* bit is set in the mbuf m_flags field.

Another portion of the mbuf is the *struct m_ext,* which is used to describe data buffers external to the mbuf structure. We add an *(*uiomove_f)()* field so that the driver can communicate a buffer- or driver-specific copy routine to the socket layer. Socket layer usage of the standard pre-existing uiomove routine assumes that the received data is in the address space and should be moved by CPU byte-copying. The indirection allows the data to be moved by programmed I/O or DMA. The uiomove_f field is valid if an M_UIOMOVE bit is set in the mbuf m_flags field. Parameters to the uiomove_f function are an mbuf, an offset into the packet at which to start copying bytes, a number of bytes to copy, and the standard uio structure that describes where the application wants the data.

The UDP input routine performs protocol processing on received UDP packets. Before the pseudo-header is constructed for checksum verification, the M_PROTOCOL_SUM bit is tested in order to skip CPU-based checksumming.

```
if (m->m_flags & M_PROTOCOL_SUM) {
  NETIO_COUNT(rch_hw_sum);
  assert(m->m_flags & M_PKTHDR);
  if (ui->ui_sum != m->m_pkthdr.protocolSum) {
    NETIO_COUNT(rch_hw_sum_bad);
    goto badsum;
 }
 goto ok;
}
```

Error processing can be based on packets reformatted into kernel buffers. The UDP output routine performs protocol processing on transmitted UDP packets.

Checksum overhead avoidance is similar to the receive path; but instead of testing the M_PROTOCOL_SUM bit, the mbuf checksum field is assumed to be valid for all transmit mbufs referencing adapter buffers (they have the M_UIOMOVE bit set). We assume that no adapter which saves the operating system the effort of data copying would forget to save the operating system the effort of checksumming. It does not make sense to eliminate some, but not all, of the per-byte overhead operations.

For UDP transmission, software recycles (adapter) buffering after the packet has been transmitted.

Changes like checksum avoidance are based on adding a conditional to the existing code paths. For a DART adapter, the test and branch penalty are small relative to the gain. For large external buffers, there are one or two M_PROTOCOL_ SUM tests per packet, depending on packet length and buffer size. This could be viewed as a constant-time overhead. The gain is avoiding the linear-time access of each byte within each packet.

For a traditional adapter, the test and branch represent overhead for each packet. The cost of the added conditionals occurs in the context of a large code base between the system call interface and the driver, and that networking code provides a rich feature set through the use of conditionals. If the added conditionals are viewed as significant, consider the approach of generating two binary files from a single source module. To avoid penalizing systems populated solely with traditional adapters, operating system software configuration procedures can choose not to incorporate the DART-conditionalized version of the code. A DART adapter installed at a later date would still operate under such a software configuration, but would not reach its peak performance until the software is reconfigured to use the DART-conditionalized version.

TCP

The TCP input routine performs protocol processing on received TCP packets. Before the pseudo-header is constructed for checksum verification, the M_PROTOCOL_SUM bit is tested in order to skip CPU-based checksumming. The only differences with the UDP input processing change are the names of the TCP header structure and TCP header checksum field.

All the adapter resources represented by the second mbuf of a received packet are consumed until the final reference to the packet is freed. If large packets are exchanged and the application is doing small reads, not until the final read is any storage reclaimed. This space consumption is represented on the socket receive queue, and therefore affects the advertised TCP window.

The TCP output routine performs protocol processing on transmitted TCP packets. The checksum overhead avoidance is similar to that done for UDP. Checksum computations for transport-layer retransmissions are simplified by the association of checksum contributions with mbufs, rather than an association of checksums with packets. The association with buffers instead of packets also simplifies handling of packets using a mix of kernel and adapter buffers.

For TCP transmission, software recycles (adapter) buffering after the packet has been acknowledged by the remote end of the connection. Between transmission and acknowledgment, the data is held on the socket's send queue. Previously, the socket code copied data from one mbuf into another whenever both mbufs' contents fit into one, trading increased CPU load for space efficiency. For DART adapters, the copy decision is cut short.

We add a *bytesSummed* field to the mbuf so that when a packet is transmitted or retransmitted by the transport layer, code can double-check that all the data the checksum is supposed to cover is still present in the

buffer. For example, a TCP acknowledgment of part of an original packet generally leads to the sender deleting its copy of the acknowledged data retransmitting the rest. The software implementation handles the generality of acknowledgments which are not complete transmit mbufs, the unit covered by the protocolSum field. A retransmission must not send a packet with an improper transport-layer checksum, even if it means using an algorithm linear in the number of bytes remaining in the buffer to recompute the checksum.

The transmitter's socket layer buffers data in segments convenient for both the network-layer protocol and the driver. Checksum contributions remembered for retransmission are recorded at a similar level of granularity. The transmitter is liberal in what the receiver can acknowledge; the receiver's implementation affects efficiency, but not correctness.

Socket Data Movement

The copy from the network buffers to the application data space occurs in the *soreceive* routine, which uses information left in the mbuf by the device driver. The call(s) to uiomove become conditionalized as follows:

```
if (m->m_flags & M_UIOMOVE) {
  assert(m->m_flags & M_EXT);
  error = (*m->m_ext.uiomove_f)(m, moff, len, uio);
}  else
  error = uiomove(mtod(m, caddr_t) + moff, len, uio);
```

The reverse copy in *sosend* is similar.

The standard uiomove function makes the optimistic assumption that the addresses of user buffers provided by the application are valid. If addresses are not valid, a trap occurs and situation-specific code is called.

To support drivers that use programmed I/O movements with the application's buffer, an additional code point is added to the error processing so that an EFAULT error is returned to the application.

Note that the changes are generic, and can be used with existing devices. The uiomove_f function can perform both copies to kernel buffers and protocol checksumming for transmission over traditional adapters.

In the transmit portion of the socket layer, the application data is moved to kernel buffers or to adapter buffers by sosend. In order to take advantage of DART adapters, sosend needs to know:

- That the protocol layers between the socket and driver support DART-style buffering
- That the driver supports DART-style buffering

In general, formatting data efficiently for transmission can require knowing the amount of headers that will be prepended by the various layers below the socket layer, so device alignment restrictions can be met. Due to protocol options and to the variety of media in existence, the amount prepended may vary from socket to socket. Given a socket, we introduced a function that computes:

- A function pointer for allocating adapter-based buffers
- A function pointer for moving data from user buffers to adapter buffers
- The number of bytes required to prepend all headers

To simplify the prototype implementation effort, the function disallows the use of adapter buffers for IP multicast packets.

When allocating adapter buffers, sosend uses the *if_xmtbufalloc* entry to allocate adapter buffers. Each time it does so, it passes a maximum number of bytes of buffering that attempts to allocate a buffer for the entire (remaining portion of the) packet. The driver indicates the actual amount of buffering allocated; sosend loops until all the necessary buffering is allocated. The driver may decline to allocate an adapter buffer if the requested amount of buffering is small. At that time the driver can best decide if CPU-based byte copying from user buffers to kernel buffers, and also copying kernel buffers to the adapter, is preferable to programmed I/O or DMA from user buffers.

Once an adapter buffer allocation fails, no further allocations are attempted within a segment that will be passed to the lower layers. This ensures that drivers will see, at worst, an (internal) mbuf containing headers, one or many adapter buffers containing data, and potentially one or many kernel buffers containing the rest of the packet. This simplifies the driver, and ensures that alignment restrictions are met without shuffling data around on the adapter. It also simplifies transport-layer checksum computation algorithms.

There is an unusual boundary case in which a long segment of transmit data may not immediately be copied to adapter buffers, even though the driver would prefer to do so. If the driver has many free transmit adapter buffers when the socket code starts to prepare a segment, it may not have any free buffers when the segment nears completion. This is because the socket layer runs at a lower interrupt priority level than the device driver, and buffers are allocated individually. A device interrupt can lead to servicing the device output queue, consuming adapter buffers in order to transmit traditional kernel-buffered packets. Rather than block and wait for transmit adapter buffer availability, the prototype software uses kernel buffers.

Both the socket and network protocol (TCP) layers contain segmentation algorithms. In the socket layer, the segmentation process is confused with the (cluster mbuf) buffer choice decision procedure. As part of eliminating that confusion, we introduce an *if_buflen* field to the ifnet structure.

If the socket layer creates segments longer than the device frame size, excess work occurs in the lower layers (e.g., TCP segmentation or IP fragmentation). If the socket layer creates segments shorter than the device frame size, the system foregoes large packet efficiencies. A large 8-kbyte write that leads to eight 1-kbyte cluster mbufs being individually processed by the lower layers might benefit from overlapped I/O of the first segment with computation of the last, but the CPU would be wasted for a benefit that is only relevant when a large number of such poorly chosen segments are constructed. Such a write could go out as a single packet over an ATM network.

Socket Buffering and Flow Control

A number of papers have commented on the requirement for a reasonable amount of socket buffering to enable applications to "fill the pipe" with a "bandwidth times delay" amount of data.[1] Delay includes the link distance, device interrupt latency, software processing, and I/O queuing delays. It also includes interrupt delays that aggregate events for efficient software processing.

The requirement for sufficient socket buffering is a lesson learned over and over again. Traditional solutions include marginal increases in systemwide defaults, and application modification to request more buffering than the default. Facilities like rsh imply that anything can become a network application, unbeknownst to the application author; so changes to applications are a poor solution. Also, applications are insulated from the network by the network protocol and socket abstractions; no application should need to know the buffering requirements for high throughput for the media *du jour.*

We introduce an *(*if sockbuf)()* entry that allows the driver to increase socket buffering. When local network-layer addresses are bound to socket connections, an interface is associated with the connection, and the driver is allowed to adjust the socket buffer quota.

For TCP server connections, the server may not be restricting incoming connections to a particular interface. Overriding the default buffering value must be done on the socket created when the incoming SYN arrives, not on the placeholder server socket. The buffer allocation needs to be determined as soon as possible, because the initial SYN packet also triggers the determination of the proper window scaling value.

UDP does not queue packets on the socket send queue. Although calls to *if_sockbuf* from the socket layer are independent of the protocol, the buffer quota only affects the maximum UDP packet size sent, not the number of UDP packets that can be in flight at the same time. The socket is not charged for UDP packets queued on the driver output queue or UDP packets in the hardware transmit queues.

The adapter buffer resources are distinct from main memory mbuf and cluster resources. The socket data structure and support routines support consumption and quota numbers for adapter buffering that are distinct from the current main memory consumption and quota numbers. For example, a connection redirected from a DART adapter to a traditional adapter is quickly flow-controlled in the socket layer as a result. The large adapter buffer allocation does not enable it to hog main memory buffers and adversely affect other connections.

IP

The prototype software contains conditionals to enable or disable the use of adapter buffers for messages undergoing IP fragmentation. This only affects UDP, since the socket layer segments appropriately for the TCP and driver layers. Software computes the amount of header space for the first fragment, and also the amount of header space for the following fragments (which will not contain transport protocol headers). This information is used during the socket layer's movement of application data to kernel or adapter buffers. UDP and IP receive the segments as a single message; the IP fragmentation code uses the fragment boundaries precomputed in the socket layer.

IP reassembly of received adapter-buffered packets was implemented in the prototype code to keep up with a transmitter using adapter buffers for IP fragmentation. The driver adjusts the hardware-computed checksum to ignore the contribution to the hardware sum caused by the successive IP fragment headers, which are not presented to the transport layer.

Resource Exhaustion

The hardware provides a scalable data memory. The memory holds received data until the application accepts it, and transmits data until the acknowledgment arrives. The prototype provides 16 Mbytes, which was considered a significant quantity after examining network subsystem buffering at centralized servers for several large "campus" sites.

When adapter memory is scarce, it should be allocated to connections whose current data flows are high-bandwidth flows. Low-bandwidth connections, connections blocked by a closed remote window, and connections over extremely loss-prone paths will not be significantly impacted by the copying overhead associated with the use of kernel buffers.

Data Relocation

Reformatting data from adapter buffers to kernel buffers allows existing code to be ignorant of adapter-buffered data. Socket-based TCP communication can use adapter buffers for high throughput, and other

protocol environments can simultaneously use the familiar kernel buffers. DART support can be phased in by protecting legacy code with a conditional relocation call before entering or queuing data to the legacy code. Cache fill operations should be targeted to main memory, not adapter memory, for best performance in legacy code.

Relocation is also appropriate for error handling and other rarely executed code paths. For example, a multi-homed host may lose TCP connectivity through the first-hop router associated with a DART link, and be forced to send packets over another link. The new communication path could use any network interface, DART or otherwise. The software needs to be able to handle the scenario where the new adapter, or some system resource, has a constraint preventing it from transmitting packets located in DART memory.

We selected a lazy evaluation solution which assumes that data sent over an old route will be delivered and acknowledged. An eager solution would incur a large burst of data relocation when the new route takes precedence, with the disadvantages that the work would be wasted for data which is acknowledged, and the burst of activity consumes resources and incurs increased latency for other activities.

For TCP connections marked as using adapter buffers, a driver entry through *(*if_ pktok)()* allows the driver to comment on each outgoing packet. This implies that the driver also comments on TCP retransmission packets. The driver has a chance to double-check constraints and trigger data relocation, if necessary. Drivers not supporting *if_ pktok* always trigger data relocation, and also lead to unmarking the TCP connection.

Comparison to Other Methods

Traditional adapters contain minimal onboard memory and hide their buffering from the CPU. Unable to manage a traditional adapter's buffers, a copy of data must be kept in host memory until it is acknowledged in case it needs to be retransmitted.

We felt copy-on-write approaches to using a traditional adapter would be inadequate due to bookkeeping overheads experienced by other projects. Also, the application may commonly reuse the same application buffer before the transport protocol semantics allow. For an unmodified application, this would lead to blocking the application, or incurring both copy-on-write and data copy overheads. All applications are network-based when one considers networked file systems and pipes to remote program invocations; architectures that require applications to be recoded to interact with page mapping schemes (e.g., [8]) are inadequate. Another objection is that copy-on-write focuses on packet transmission, ignoring packet reception.

When a write is performed by an application using DART, the application blocks only long enough to buffer the data, as for a traditional adapter. The copy of the application's data on DART enables retransmission for reliable communication. The application is free to immediately dirty its write buffer, and no performance impact is associated with that action.

Van Jacobson's WITLES paper design uses the CPU to copy data to and from the adapter via programmed I/O.[9] Reading the adapter is an expensive operation, and in practice would provide worse receive performance than even a traditional adapter. The Medusa design is a WITLES variant that uses programmed I/O transmission and addresses the receive penalty with system block-move resources for reception.[10] The Afterburner design used the same approach, achieving 200 Mb/s.[4] The WITLES approach keeps the packet in adapter memory until it is copied to the application buffer.

To minimize resource consumption, the checksum and copy loop are combined. This means that the TCP acknowledgment is deferred until the application consumes the data, which might be much later than necessary. Applications read data at a rate of their own choosing. Care must be taken that this deferral does not lead to TCP messages to the data source that cause unnecessary data retransmission.

Unlike WITLES, DART supports DMA to and from the adapter. Software can use DMA where appropriate, intelligently balancing the costs of programmed I/O and DMA.

Since DART provides the IP checksum with the packet, the TCP acknowledgment can be sent as soon as the packet is reassembled and reported to the CPU. The acknowledgment contents and transmission time are traditional BSD UNIX; it states that the data has been received, and the offered window reflects buffer consumption until the application receives the data at its leisure.

Adapters have been built that offload protocol processing.[1] However, the cost of TCP processing is low, and such an architecture introduces message-passing overheads that counterbalance the offloaded protocol processing efficiencies. CPU execution rates are scaling well. The issue to address is the main memory bandwidth bottleneck. Also, it is expensive and difficult to create, maintain, and augment the firmware for such an adapter. The firmware is tied to a single adapter, and replicates work done within the operating system that can be shared by a number of adapters.

DART provides assist via checksumming methods. It does not attempt to offload network- or protocol-layer processing.

Performance

The simulation environment used to debug and test the chip design was also used to extract performance

information. The chip model used to fabricate the part is connected to a PCI bus simulation, some generic bus master devices, and some generic bus slave devices. The simulation environment is connected to and controlled by a TCL-based environment.

Within the TCL environment, the hardware designers wrote a device driver. With this driver, DART copied packets from host memory, looped packets on an external interface, reported packet summaries, and copied packets into host memory. Both 64- and 32-bit PCI buses were exercised. Target read latency of host memory was incorporated into the simulation (the data presented in Figure 7 is based on a 16-cycle latency). Credit-based flow-control operations were enabled since they consume additional control memory bandwidth, and therefore represent worst-case-scenario operation, Similarly, a large number of virtual circuits were used to loop data, to prevent the use of on-chip, cached circuit state.

Because the TCL driver was written by hardware designers, and they were focused on designing and testing the chip, performance numbers extracted from their work suffer from a lot of CSR accesses. A real driver would reduce the CSR operations and have increased batching of interrupts and other actions.

CSR reads are costly, since they involve a round-trip time within the chip which crosses clock boundaries, in addition to the round-trip time between the CPU and the pins on the device. Crossing clock boundaries means that there are internal first-in first-out (FIFO) delays involved to deal with synchronization and meta-stability issues. To meet PCI latency specifications, the bus master is told to retry such operations, freeing the PCI bus for other use during the internal round-trip time. CSR writes are efficient, since they are buffered throughout the levels of the system.

The dip in Figure 7 is near the 512-byte burst size used to read from host memory. Packet transmissions no longer fit in a single DMA burst, and incur the extra cost of an additional short fetch. This incurs additional overhead cycles to place the address on the bus and for the target to start to respond with the first bytes.

For each simulation we extract numerous detailed statistics. Table 3 contains a few for 32-cell packets (1536 bytes) on a 32-bit PCI bus. These particular figures are for the TCL driver, and include time intervals to initialize the adapter, to transmit before the first packets are received, and to receive after the last packet was transmitted.

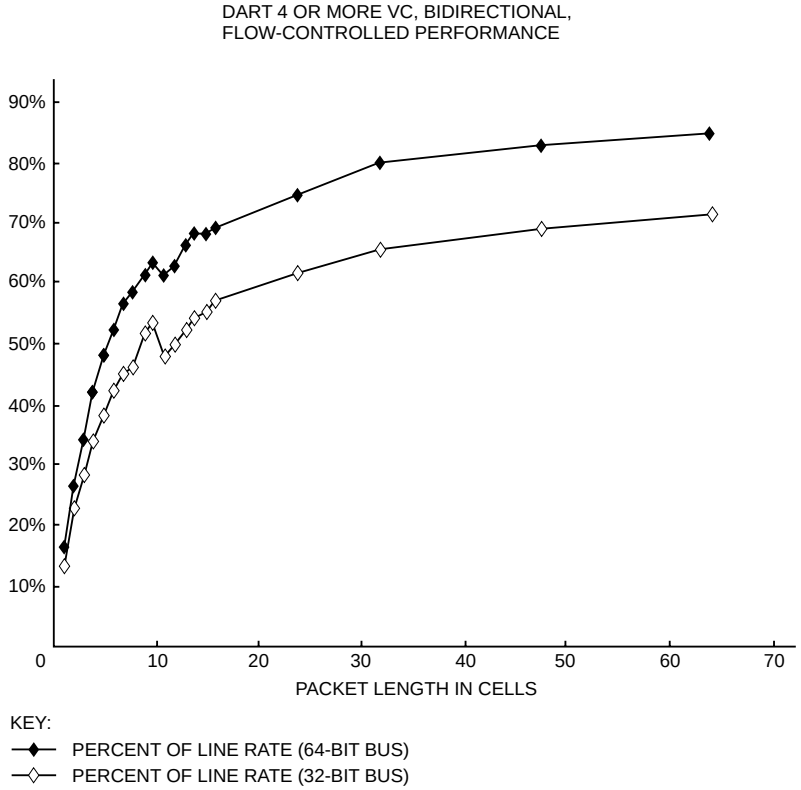

Figure 7
DART Performance

Table 3
Examples of Additional Statistics

Control memory idle	79%
Data memory idle	48%
PCI busy (frame or irdy asserted)	75%
PCI transferring data (irdy and trdy asserted)	60%
CSR operations share of bus operations	41%

Future Work

Due to the large amount of onboard buffering, we do not expect DART to encounter resource exhaustion issues. However, some work will be appropriate to determine the best solution should buffering requirements exceed the electrical capabilities of the high-speed SAR-SDRAM interface. Is it efficient to move unacknowledged data off the adapter so that new transmit data can be moved from user space to the adapter in the socket layer? Is it efficient to block in the socket layer, waiting for adapter buffers to be freed by a future, or arrived but unprocessed, acknowledgment? Is it efficient to use conventional kernel buffers to transmit when the space allocated to DART-style transmissions is exhausted?

DART structures the system software so that the operating system does not examine the application's data, which should be private to the application anyway. This separation of control operations (on headers) from data operations (primarily movement) is a common theme in embedded system design for bridges and routers. DART provides a generic structure that enables high-performance networking in a variety of systems.

With features like peer-to-peer I/O, one can conceive of a system with multiple gigabit links, where the bottlenecks have shifted from the system software to the application or service. We think DART-style adapters will enable and accomplish the delivery of high-bandwidth service to the application.

Acknowledgments

Robert Walsh implemented the transmitter and PCI bus interface. Kent Springer implemented the receiver and packet reporting functions. Steve Glaser implemented the DMA engine. Tom Hunt implemented the external control RAM interface, the external data RAM interface, and the board design. Robert Walsh developed the prototype UNIX changes. Phil Pears, Mark Mason, James Ma, and Ken-ichi Satoh provided significant assistance in placing and routing the ASIC.

We also had assistance from Joe Todesca, Elias Kazan, and Linda Strahle. Bob Thomas participated in the initial concept and design. K.K. Ramakrishnan provided some information on networking performance.

References

1. Metcalfe, "Computer/Network Interface Design: Lessons from Arpanet and Ethernet," *IEEE JSAC,* vol. 11, no. 2 (Feb. 1993).
2. Walsh and Gurwitz, "Converting the BBN TCP/IP to 4.2BSD," *USENIX 1984 Summer Conf. Proc.* (June 1984).
3. Chang et al., "High-Performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP," *Digital Technical Journal,* vol. 5, no. 1 (Winter 1993).
4. Dalton et al., "Afterburner," *IEEE Network* (July 1993).
5. Clark et al., "An Analysis of TCP Processing Overhead," *IEEE Commun. Mag.* (June 1989).
6. Kay and Pasquale, "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000," *USENIX 1993 Winter Conf. Proc.* (1993).
7. Owicki, "AN2: Local Area Network and Distributed System," *Proc. 12th Symp. Principles of Dist. Comp.* (Aug. 1993).
8. Smith and Traw, "Giving Applications Access to Gb/s Networking," *IEEE Network* (July 1983).
9. Van Jacobson, "Efficient Protocol Implementation," ACM SIGCOMM 1990 tutorial (Sept. 1990).
10. Banks and Prudence, "A High-Performance Network Architecture for a PA-RISC Workstation," *IEEE JSAC,* vol. 11, no. 2 (Feb. 1993).

Additional Reading

1. Kay and Pasquale, "The Importance of Non-Data Touching Processing Overheads in TCP/IP," *Proc. SIGCOMM '93 Symp. Commun. Architectures and Protocols* (1993).
2. Ramakrishnan, "Performance Considerations in Designing Network Interfaces," *IEEE JSAC,* vol. 11, no. 2 (Feb. 1993).

Biography

Robert J. Walsh
Robert Walsh has been working on high-speed networking since the beginning of the 1980s. He developed networking software for BSD UNIX, BBN's Butterfly multiprocessor, and DIGITAL's GIGAswitch/FDDI.