MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

DART — A Low Overhead ATM Network Interface Chip

Randy Osborne, Qin Zheng, John Howard, Ross Casley, Doug Hahn, Takeo Nakabayashi

TR96-18 December 1996

Abstract

DART is an ATM Network Interface Controller chip designed for both high bandwidth and low overhead communication. Innovative features are direct, protected application access to/from the network, host-assisted flow control, and "message processing" for flexible low overhead communication mechanisms. DART is being manufactured as a commercial product by Mitsubishi Electric.

Hot Interconnects IV: A Symposium on High Performance Interconnects, August 1996, pp. 175-186

MERL – A MITSUBISHI ELECTRIC RESEARCH LABORATORY
http://www.merl.com

DART — A Low Overhead ATM Network Interface Chip

Randy Osborne, Qin Zheng, and John Howard – MERL
Ross Casley and Doug Hahn – Sunnyvale Research Lab
Mitsubishi Electric Information Technology Center America, Inc.

Takeo Nakabayashi – System LSI Laboratory, Mitsubishi Electric Corp.

TR-96-18 July 1996

Abstract

DART is an ATM Network Interface Controller chip designed for both high bandwidth and low overhead communication. Innovative features are direct, protected application access to/from the network, host-assisted flow control, and "message processing" for flexible low overhead communication mechanisms. DART is being manufactured as a commercial product by Mitsubishi Electric.

In proceedings of *Hot Interconnects IV: A Symposium on High Performance Interconnects*, Stanford University, Palo Alto, California, August 15–17, 1996

1. First printing, TR96-18, July 1996

DART*— A Low Overhead ATM Network Interface Chip

Randy Osborne†, Qin Zheng, and John Howard – MERL
Ross Casley and Doug Hahn – Sunnyvale Research Lab
Mitsubishi Electric Information Technology Center America, Inc.

Takeo Nakabayashi – System LSI Laboratory, Mitsubishi Electric Corp.

Abstract

DART is an ATM Network Interface Controller chip designed for both high bandwidth and low overhead communication. Innovative features are direct, protected application access to/from the network, host-assisted flow control, and "message processing" for flexible low overhead communication mechanisms. DART is being manufactured as a commercial product by Mitsubishi Electric (part no. M65433).

1 Introduction

The DART ATM Network Interface Controller integrates ATM adaption layer processing (AAL5 and AAL0) and a PCI bus interface into a single chip for full duplex 155Mbps ATM local area networks. DART permits high performance — both high bandwidth and low latency — via hardware that supports direct, protected, application access to the network. While such operating system bypass techniques have been implemented in prototypes [1,3,5] (sometimes for special purpose networks) or in firmware using expensive NIC cards [12], to our knowledge DART is the first commercial application of this technique to a NIC chip for mainstream LANs. The one way latency for a small message from application to application is 11μsec without a switch and is 16 to 35μsec with a single switch, depending on the switch and physical medium.

The reason for our interest in low overhead communication is two-fold. First, small messages are important. In today's client/server dominated computing systems there are many small request/response messages. The throughput for such small messages is dominated by host overhead and round trip latency. Second, low overhead communication is important for emerging ways of building distributed systems, particularly cluster-based computing. Predictable low latency is also important for real-time computing requirements, such as in industrial control systems.

For very low overhead communication, DART supports "message processing" — low level processing of AAL5 and AAL0 frames between the network and application. Possible uses include low overhead communication mechanisms (e.g. sender-based protocols [3,9], remote queues [2], and remote memory operations — read and write of memory in other computers), reflective memory in real-time systems, filtering of unwanted messages, and fast/cheap demultiplexing.

DART is a fully featured NIC chip intended to satisfy the demands of the commercial market. DART's traffic management supports both the ATM Forum rate-based flow control scheme [11] and the Quantum Flow Control Consortium credit-based flow control scheme [8] with flexibility to follow standards evolution and implement proprietary schemes. The transmit side implements per connection traffic shaping capable of any transmission rate, four transmission priority levels; and multiple traffic classes (CBR, VBR, ABR, UBR). DART supports up to 32K transmit virtual channels (VCs) and 64K receive VCs active simultaneously. To provide a low cost solution, DART can use host memory for data packets, includes full support for EDO-DRAM local memory, and does not require a local processor.

Figure 1 shows the DART chip and an example

*DART (Direct Access Receive and Transmit) is a trademark of Mitsubishi Electric Information Technology Center America, Inc.

†Contact: osborne@merl.com or 201 Broadway, Cambridge, MA 02139

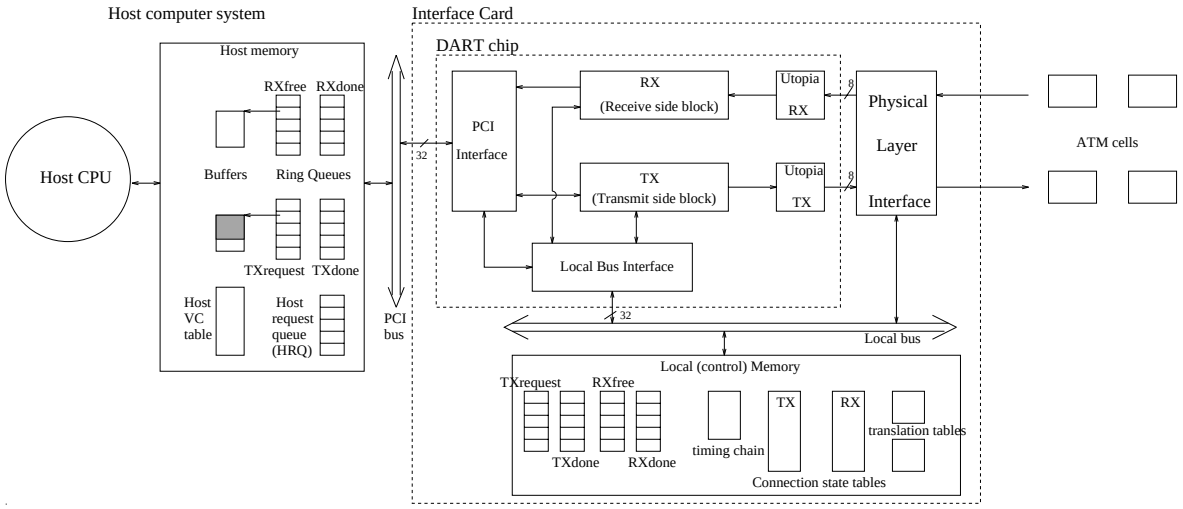

Figure 1: DART chip and example operation mode

operation mode. Local memory contains per-VC state and a timing chain for traffic shaping. Communication between host and DART is via ring queues: TXrequest and TXdone are transmit request and transmit complete ring queues; RXfree is a free buffer ring queue; and RXdone is a receive complete ring queue. All ring queues and data buffers can either be in host memory or local memory. DART requests special host action via the host request queue (HRQ). The host VC table holds per-VC traffic management state.

DART's PCI bus interface incorporates a 64 word write buffer for transfers to host and a 32 word buffer for host writes to DART. Cells are sent and received via separate UTOPIA physical layer interfaces (8 bits wide) with a 39 cell receive FIFO and a 4 cell transmit FIFO. To reduce design complexity DART has sole mastership on the local memory bus. The local bus interface supports 4 banks of DRAM memory with all necessary control logic and also 4 banks of SRAM and/or general external devices.

2 Direct Application Access

There has been much research work documenting the overhead of conventional kernel-based network interfaces and investigating OS bypass techniques (e.g. [1,5,10]). The simple design of conventional network interface hardware forces kernel demultiplexing, kernel calls, and heavyweight software interfaces (a one size fits all approach).

To support high bandwidth, low overhead, and application-specific protocols, DART allows an application to bypass the OS by providing direct, protected access to the network. DART uses the Application Data Channel [5] idea: hardware looks up transmit privileges and demultiplexes received messages using the VC number to index into state tables maintained by the OS. Address translation hardware ensures that applications only send and receive from designated areas setup by the OS.

Figure 2 shows DART's two direct application access (DAA) channels. Each DAA channel has a full set of ring queues, and other mechanisms and data structures to interact with an application. The ring queues and transmit and receive buffers can be mapped into application space. The transmit side ensures that an application can only send cells into the network for the VCs assigned to that DAA channel. The receive side routes messages to the proper DAA channel. Each DAA channel has a separate two-way set associative, variable sized address translation table implemented using NIC local memory and managed by the OS. We expect that typically one DAA channel will be reserved for the OS to use as in a conventional network interface. Each DAA channel has two free buffer queues, allowing different buffer areas for different VCs sharing a DAA channel. This permits a VC to choose the buffer characteristics e.g. use one free queue for VCs with small messages,

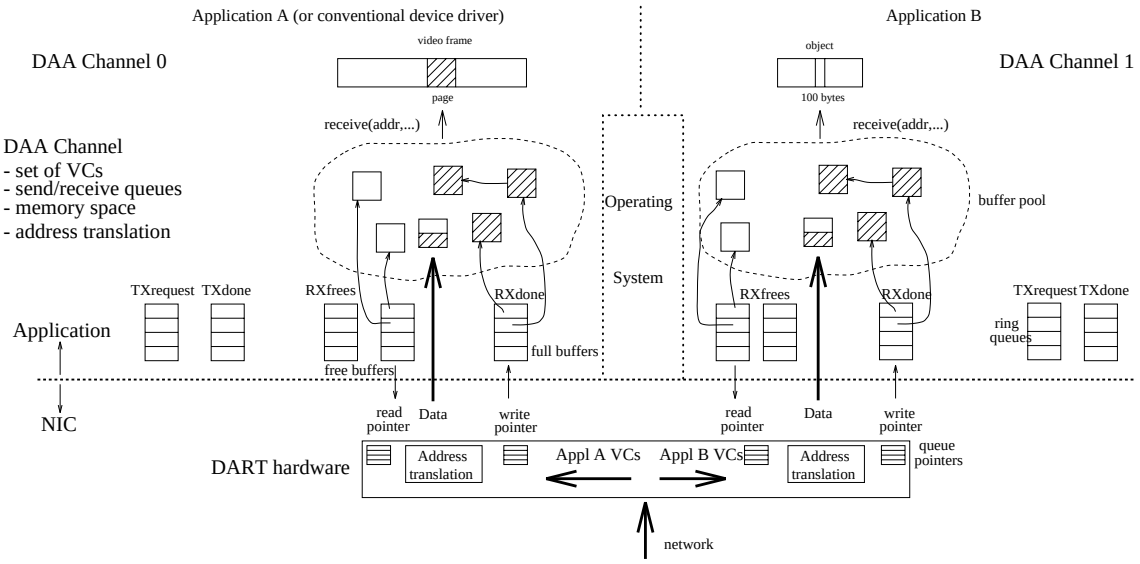

Figure 2: Direct access architecture, receive side emphasized

another for VCs with large messages. Two free queues also allows some VCs to use credit-based traffic management and other VCs simultaneously to use rate-based.

The ring queues contain fixed size frame descriptors (FDs) shown in Figure 3. The transmit side has both a short form frame descriptor (FD) format to reduce the overhead for short messages and a long form frame descriptor format for larger messages. The long form allows multiple frames to be chained together as described in Section 2.1. The receive side only uses the short form for both free buffers and filled buffers. The 4-word FDs allow everything describing a short frame to be conveyed in a single PCI bus transaction. The presence bits in each FD allow the host to poll output queues like RXdone and DART to poll input queues like TXrequest. Dual presence bits (marked P in the first and last words in an FD) allow each ring queue to either be in host memory (desirable for output queues) or local NIC memory (desirable for input queues). To notify DART of a non-empty TXrequest or to initiate polling of a TXrequest, the host writes into a notify_in register: the value written indicates the number of times to read TXrequest before stopping. To allow low overhead notification, the notify_in register (for DAA Channel 1) can be mapped into application space to enable direct writing by the application.

The following subsections describe novel aspects of DART and implementation details.

2.1 Dynamic Chaining

Head of line blocking can arise in TXrequest if the head entry cannot be dequeued because the VC is already busy sending a frame and entries for higher priority VCs follow that head entry. One way that others, e.g. Texas Instruments' 1561 PCI NIC, have solved this problem is to have a queue per VC.[1] Beside the obvious scalability problem with the queues, there is a more subtle problem dealing with notification. DART must know which ring queue to read; it cannot simply poll all the queues. A bit vector with one bit per queue is appropriate for notifying up to perhaps 64 or 128 queues, well short of our design goal of 32K VCs. To scale to that range requires some way to indicate a queue number to DART and that implies another queue just for notifications. To allow scalability to 32K VCs we decided to use one queue (per DAA Channel) and folded the notification into the TXrequest queue.

To solve the head of line blocking problem we chose a novel solution that dynamically chains together long format frames. Long frames are comprised of a linked list of buffer descriptors (BDs).

[1] Or at least per class of request. However, since each ABR VC is independently flow controlled, each ABR VC must belong to a different queue.

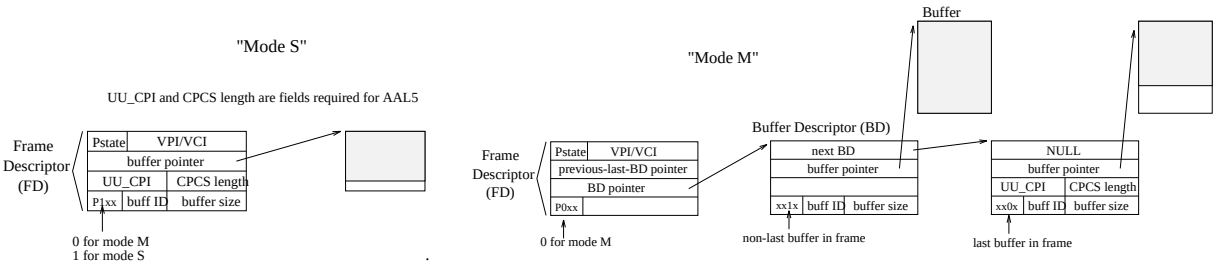

Figure 3: Short and long frame formats

The FD enqueued for frame i of VC j contains a pointer (called the *previous-last-BD pointer*; see Figure 3) to the last BD in frame $i-1$ for VC j (the driver or application can record this). If DART reads such an FD from TXrequest while VC j is busy, it writes the BD pointer contained in the FD of frame i into the address specified by the previous-last-BD pointer in the FD, thus chaining together the BDs for frames $i-1$ and i. Dynamic chaining pertains only to long mode frame formats. Short form formats can cause head of line blocking and thus must be used carefully by the driver/application (but yield low overhead).

The driver/application can implement the following simple algorithm to dynamically choose how and when to enqueue a frame in TXrequest. For each VC maintain a count, `#frames`, of the number of frame segments for that VC channel enqueued in TXrequest but not yet seen in TXdone. Assuming that `#frames` is initialized to -1 (the "credit"), enqueue a frame of whatever type if `#frames` is -1 and convert short mode frames to long mode frames if `#frames` ≥ 0. If `#frames` is sufficiently large, it is more efficient for the driver/application to statically chain the frames together and then enqueue the entire chain when `#frames` drops to some smaller value.

2.2 Chunking

To reduce the internal fragmentation in the use of free buffers for receiving short messages, DART allows multiple messages to be stored in a buffer. The next message for a VC is stored into the remaining buffer chunk following the previous message for that VC. This solution provides better buffer efficiency than the conventional solution of dynamically choosing between a small buffer queue and a large buffer queue since it adapts dynamically to the buffer size messages require. Another reason for chunking is to enable easy implementation of credit-based traffic management schemes: credits can be forwarded based on the total buffer size in the free buffer queue, without worrying about buffer space unused due to fragmentation.

2.3 DAA Scalability

With one DAA channel devoted to a traditional driver interface, only one application can exploit the direct access and low overhead of the remaining DAA channel. It is expensive (in hardware) to support more physical DAA channels. DART provides two ways to scale DAA to more applications with some degradation in performance, though much less than traditional approaches.

The first way is RX side per-VC buffering: each VC may have a private set of buffers more tailored to the requirements of that VC. This set is implemented as a linked-list of BDs with the head pointer in the RX VC table entry. When the VC exhausts this list, it reverts to a RXfree buffer queue. While not completely equivalent to a DAA Channel, this does allow low latency messages to be deposited directly to application memory even if the RXdone ring queue is shared by multiple applications.

The second way is place the ring queues in kernel space so multiple applications can share them without protection problems while placing the data buffers in application space for direct access.[2]

More specifically, on the TX side TXrequest and TXdone can be in kernel space and data buffers and BDs in application space. This means an OS operation in the critical path of sending in order to

[2]Address mapping can be configured independently for the various ring queues and data buffers.

enqueue a FD and validate pointers. However, as discussed in Section 2.1, some form of serialization is required for transmission notification. The simplest way to meet this requirement is to leave the TXrequest queue in the kernel. Each application only needs to export that portion of its virtual address space corresponding to the buffers and BDs it wishes to send. On the RX side, one or both RXfrees can be in kernel space, and RXdone and data buffers in application space. Since applications only read RXdone, it can be in shared space where it can be accessed quickly via each application for low overhead. However, the RXfree queues, like TXrequest, must be writable, and hence must be in kernel space for protection between multiple applications. Each RXfree can support a buffer pool with a different virtual address space range by allowing the entries in the RXfree queues to contain virtual addresses. The applications or kernel can add buffers to these pools and the driver checks that the buffer address space ranges are appropriate for the given RXfree queue.

2.3.1 Virtual DAA Channels

A better approach to provide DAA scalability is to virtualize the DAA channels. We investigated the following hybrid virtualization strategy to the point of RTL implementation and testing, but dropped it from the final design for business and schedule reasons. As mentioned above, the simplest way to scale the TX side is to leave the TXrequest queue in the kernel. However, to ensure that the OS operation required is just a cheap kernel trap and enqueue operation (and no pointer validation), a field in the TX VC table gives the index of an appropriate translation table to use when translating buffer and BD pointers for that VC. The appropriate translation table is dynamically loaded on demand.

On the RX side, all the resources are virtualized for DAA channel 1. Up to 4K RXfree and RXdone ring queues can be supported by multiplexing the DART DAA Channel 1 ring queue registers amongst multiple ring queues. The ring queue pointers are kept in local memory and loaded into the DART ring queue registers on demand, like virtual paging. As on the TX side, the translation table is also virtualized. A field in each RX VC table entry contains indices for the appropriate RXfree, RXdone, and translation table to use for that VC. Before reading RXfree to get a free buffer or storing a FD to RXdone, DART loads the proper RXfree or RXdone ring queue pointers. The IN and OUT pointers are written back to the ring queue tables afterwards. Each virtual ring queue has the same full/empty and watermark interrupts interrupts as for the base ring queues, but these interrupts are delivered via HRQ entries to allow scalability.

2.4 Implementation Details

2.4.1 Translation and Fault Handling

DART translates an application virtual address — of a data buffer, of a buffer descriptor, or in a ring queue register — to a physical address before accessing physical memory. This translation also checks that any pointers (data or buffer descriptor) provided by an application are legitimate. To reduce translation overhead DART retains virtual and physical versions of most pointers and only translates either when a structure (e.g. data buffer) is first accessed or on every page crossing.

DART performs translation using a cache of translation entries implemented using a table in local memory. In addition to simplifying DART, placing the translation cache in local memory makes it easy to adjust the cache size to achieve a suitable hit rate. Placement in local memory also permitted a two way set-associative cache with little additional complexity. Two sets allows further flexibility in configuring DART to maximize hit rates. One example use is to maintain all the ring queue entries in one set so that no troublesome ring queue pointer misses result from capacity or conflict misses with buffer addresses.

The translation is quite general. DART interprets an address given by an application as a token and a page offset and examines the translation table for a mapping from that token to a physical page number. The token may be a virtual page number or it could be any other identifier for a virtual page that the application arranges (e.g. an index into a address table).

A translation miss in both sets of the translation cache causes a translation fault. DART enqueues a translation fault record in the HRQ which contains the address causing the fault and then DART interrupts the host. Thereafter faults are handled differently on TX and RX sides.

If a fault occurs on the TX side, DART freezes the associated VC (for a buffer pointer or a BD pointer fault) or freezes the entire DAA channel

(for a ring queue pointer fault). The host either fixes the fault and issues a thaw command to resume operation or issues an abort command to abort the frame (e.g. in case of an illegal address) or shutdown the DAA channel.

Since DART has little control over the arrival of cells, it cannot freeze operation indefinitely if a fault occurs on the RX side. Instead, DART temporarily suspends operation on the RX side: it backs out of the faulting operation and then DART periodically retries the translation. If the translation succeeds within the retry window, operation continues from the faulting address. If not, there are two cases: 1) If the fault is the base address of a new buffer, DART abandons that buffer and fetches a new buffer from the appropriate RXfree queue. The RXfree is disabled if two such timeout faults occur in a row for the same VC. This limits the number of faults a malicious application can cause by loading illegal buffer addresses into a RXfree. 2) Otherwise, DART aborts the current operation, either discarding the current frame if a buffer pointer fault or shutting down the DAA channel if a ring pointer. The retry window extends for 50μsec or until the receive FIFO is 3/4 full (30 cells), whichever is smaller. If the host can fix a fault within this interval it can pin DMA pages on demand.

HRQ overflow requires careful handling. A fault record cannot simply be dropped if HRQ is full since then the host will not be able to determine that a fault occurred. Instead the operation causing the fault is rolled back and for a TX side fault, DART freezes the the VC or ring queue and for a RX side fault, DART discards the frame or disables the ring queue. Other types of HRQ entries, e.g. ABR requests (see Section 3.2) are dropped if HRQ is full.

2.4.2 Ring Queue Operation

The dual presence bits shown in Figure 3 permit ring queues to be in either host or local memory as follows. For a single presence bit scheme, the invariant required to avoid races is the producer of a FD writes the presence bit in the last word written and the consumer reads the word with the presence bit first. For simplicity DART always reads and writes FDs in increasing address order and thus for queues in host memory the host can obey the above invariant by accessing FDs in the opposite order. However for queues in local memory, the PCI bus forces host accesses in the same ascending order as DART. A second presence bit at the other end of the FD sidesteps this overconstraint by using different presence bits to meet the invariant: both presence bits must be set in a valid FD.

Using presence bits per FD plus ring queue pointer caching can reduce the average number of PCI accesses per FD enqueued or dequeued to nearly 1. The host clears the presence bits wherever the ring queue is located. For an input queue (TXrequest or RXfree) in host memory, DART maintains only an OUT pointer and the host maintains an IN pointer and NIC_OUT, a cached copy of the DART's OUT pointer. The host periodically reads OUT and clears the presence bits in queue entries from NIC_OUT to OUT-1 and then updates NIC_OUT. For an output queue (TXdone or RXdone) in host memory, DART maintains an IN pointer and a copy, HOST_OUT, of the host's OUT pointer. DART may fill the output queue until IN = HOST_OUT. The host periodically clears the presence bits in queue entries from OUT to HOST_OUT and then writes HOST_OUT-1 to OUT.

Each ring queue has a watermark interrupt, signalled when the number of entries falls below for input queues/above for output queues a parameterized level. The RXdone queues also have a special interrupt signalled when the number of end-of-frame entries exceeds a set level.

2.4.3 TX Pipelining

The TX side functions are considerably more extensive than on the RX side. Whereas the RX side simply receives cells and stores them to buffers, occasionally fetching a free buffer and occasionally writing to RXdone, the TX side must segment data buffers (i.e. send cell data), perform traffic shaping and traffic management, and fetch FDs from TXrequest. Traffic shaping and management require significant logic and a large amount of local memory state to be fetched and updated per cell time. To sustain high bandwidth, particularly for small message sizes, the TX side must be able to quickly find FDs enqueued in TXrequest and prepare the respective VCs for segmentation. Finally, the TX side must tolerate PCI latency to host memory (whereas the RX side can overlap PCI latency by using a writeback buffer).

To meet TX side performance objectives we took the following steps. First, we cached VC state ta-

ble entries, using a writeback policy. This significantly improved performance for multi-cell bursts belonging to the same VC. To improve the performance for the more general case of interleaved VCs we pipelined the segmentation. We used a single stage pipeline to overlap the PCI payload transfer with writing back VC state and starting the next cell. Finally, to lessen the impact of polling TXrequest, we overlapped the PCI read from TXrequest with the startup of the next cell. The actual processing of the FD read is delayed until after the VC state writeback. Overlapping these operations allows DART to achieve full line rate in the presence of substantial PCI bus latency.

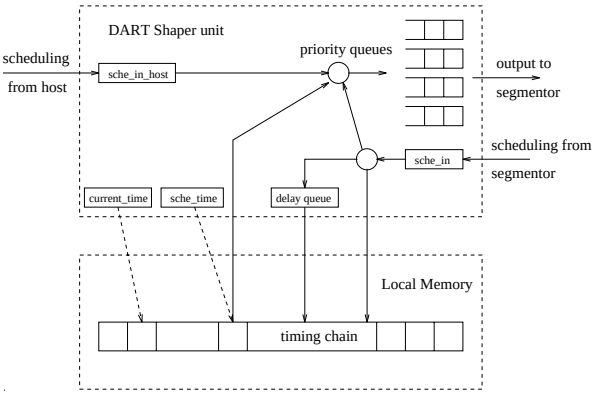

Figure 4: Traffic shaper

3 Traffic Management

DART provides functions supporting multiple service classes and allowing implementations of a wide variety of traffic management schemes including ATM Forum's TM 4.0 and the Quantum Flow Control Consortium's QFC. This provision of flexible and upgradable traffic management functions is essential to follow evolving traffic management specifications and to accommodate new flow control schemes.

Specifically, DART provides: 1) Traffic shaping per VC capable of supporting up to 32K VCs with transmission rates ranging from 155.52 Mbps to 0.3 Kbps at a resolution of 0.3 Kbps. 2) Leaky bucket traffic shaping to support variable bit rate (VBR) VCs. 3) Four transmission priority levels to support multiple service classes, including CBR (constant bit rate), rt-VBR (real-time VBR), nrt-VBR (non-real-time VBR), ABR (available bit rate), and UBR. 4) Software-assisted ABR flow control (implemented by the host processor) to support standard and proprietary ABR schemes.

3.1 Traffic Shaping

Traffic shaping is an essential function to provide quality of service and avoid traffic congestion in ATM networks. This function is required for all ATM service classes supported. ATM Forum Traffic Management Specification [11] provides a detailed description of the various ATM service classes and their traffic shaping requirements.

The traffic shaper schedules cell transmission times for each VC such that the inter-cell time dt for a VC is controlled above a certain value. In other words, after transmission of a cell, the traffic shaper schedules the VC to send another cell at a specified future time. If more than one VC is eligible to send a cell, a VC with the highest priority is selected to send a cell first. Priority control is needed to support multiple service classes such that a VC with tight delay or bandwidth requirements can be assigned a higher priority than others.

DART uses an enhanced timing chain to implement traffic shaping and priority control. This approach has the advantage of being able to traffic shape a large number of VCs at virtually any rate values. It supports both peak rate and leaky bucket shaping and integrates traffic shaping and priority control into a single mechanism.

Figure 4 shows the major elements in DART for traffic shaping and transmission priority control.

Local memory contains a timing chain for scheduling VC transmission times. Each entry represents a cell transmission time (at full line rate) and contains a list (possibly empty), of VCs eligible to transmit a cell at that time. current_time points to an entry representing the current time and moves forward one entry per cell time. sche_time points to an entry from which VCs are being dequeued from the timing chain. sche_time moves forward one entry after it dequeues all VCs in a entry until eventually sche_time catches up with current_time. To rate control a VC with a cell rate R, after the transmission of a cell the VC is rescheduled into an entry which is 1/R entries away from the entry pointed to by current_time.

The delay queue enables traffic shaping to an arbitrarily low rate. If a VC's inter-cell transmission time is too large to fit into the timing chain,

the VC is instead enqueued in a linked list delay queue. The head of the delay queue linked list is examined periodically and either transferred to the timing chain if possible or re-queued at the end of the linked list.

Each VC eligible for transmission, as specified by the current time entry, is dequeued from the timing chain and transferred to an on-chip priority FIFO which is used to schedule VCs for segmentation and cell transmission. Each VC has one of four possible priority levels. There are 8 FIFO entries for priority levels 0 and 1, 16 entries for level 2, and 32 entries for level 3.

After the segmentor transmits a cell, it reschedules the VC in the timing chain to send the next cell (if any data remains available for that VC)[3]. The segmentor indicates a value *dt* which means the VC should be scheduled *dt* entries away from the current_time pointer in the timing chain. Different types of traffic shaping algorithms, e.g., leaky bucket traffic shaping for VBR VCs, can be implemented by using different ways of calculating *dt*.

There are two ways to initiate transmission for a VC. The main way is to insert a FD in the appropriate TXrequest queue. If the VC is idle — or more precisely, that VC is not scheduled in the timing chain — when DART reads the FD, DART will insert the VC directly into the shaper FIFO and the segmentor sends the first cell of the frame as soon as possible. A secondary way to initiate transmission is by writing the sche_in_host register. Occasionally it is necessary for the host to re-activate a stopped VC (e.g. resume after flow control throttling) or send non-shaped raw cells (again, for flow control). To restart a stopped VC, the host writes the VC number into sche_in_host and that VC is moved directly to the priority FIFO queue. To insert a single cell, the host writes the base address of the cell into sche_in_host and this cell pointer is likewise moved directly to the shaper FIFO.

3.2 ABR Flow Control

DART provides unique hardware support to enable flexible and low overhead ABR flow control in software on the host system to enable a low-cost end system implementation. Both the ATM Forum's rate-based TM4.0 ABR flow control and the Quantum Flow Control Consortium's credit-based

[3]After completing a frame the VC is rescheduled once more to ensure proper spacing between frames.

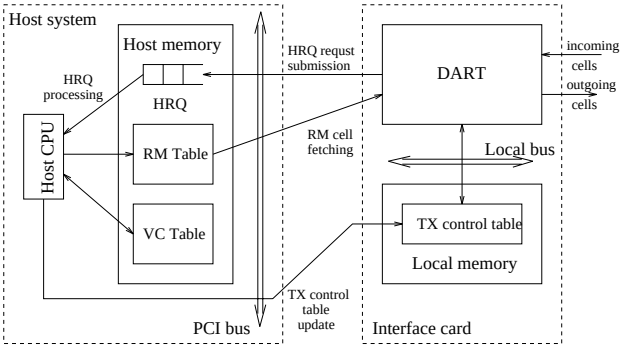

Figure 5: ABR flow control framework

lossless ABR flow control schemes can be fully implemented at the host driver level.

Figure 5 shows the framework for ABR flow control in DART. For the ATM Forum rate-based scheme, DART generates forward Resource Management (RM) cells by DMA from per-VC RM cell templates in host memory. RM cells received from the network are enqueued by DART in the Host Request Queue (HRQ) in host memory. The host periodically examines these RM cells, turning around forward RM cells by modifying a per-VC field in local memory and processing backward RM cells by adjusting the per-VC rate parameters in local memory. In addition to the forward and backward RM cell events, an entry is also enqueued in the HRQ whenever a forward RM cell is sent. This permits implementation of source timeout rate reduction.

For the QFC scheme, DART meters cell transmission according to the outstanding credit balance. To perform the cell metering, DART maintains a per VC cell counter on the TX side. When this reaches a defined cell_limit for that VC, DART halts transmission for that VC and enqueues a VC-stopped message in the HRQ. On the RX side, DART forwards all received QFC credit and protocol cells (except for state check cells) to the host via the HRQ. The RX side also maintains a per VC cell counter for state synchronization. With the exception of state check cells, the host is responsible for 1) generating (as raw cells) all credit and QFC protocol cells, 2) processing all credit and QFC protocol cells in HRQ, and 3) implementing the QFC protocol, including restarting a stopped VC when credit arrives. QFC uses state check cells to resynchronize the sender and receiver state in case of lost cells. DART generates check cells to ensure

Event	Time
interrupt entry	3.6 to 8.6
3 RM cell events (hot caches)	4.1
3 RM cell events (cold caches)	15
interrupt exit	2.1

Table 1: ABR overhead (all times in μsec)

that they have the proper cell count. On the RX side, DART maintains the proper position of the check cell within the data stream by writing a special FD to RXdone containing the information in the check cell, the RX cell count, and the current buffer offset for that VC.

DART's host assisted traffic management incurs some host overhead. On one hand, the host should be interrupted as infrequently as possible to minimize the overhead. On the other hand, ABR requests submitted to the HRQ should be processed as soon as possible to improve the responsiveness of ABR flow control. To minimize the host ABR overhead, DART generates just one HRQ interrupt for all ABR requests submitted within a user specified interrupt period.

We estimated host overhead in the target PC+Windows NT environment as follows. Since we do not yet have a full WNT driver for DART, we instead made three separate measurements: 1) We measured the interrupt "entry" time from application until acknowledging a network interface interrupt in the lowest level WNT interrupt handler. 2) At user level we measured the time to perform the 3 RM cell handling events, including reading the HRQ, that we would expect for sustained ABR traffic (source sending a RM cell, destination turning around a received RM cell, and source receiving a RM cell). 3) We measured the interrupt "exit" time from acknowledging a network interface interrupt in WNT until return to application level. To measure entry and exit times we patched the WNT binary to collect performance data using the Pentium hardware event counters.[4]

The results for a 100MHz Pentium PC with 256Kbyte secondary cache and 40Mbytes non-EDO DRAM are shown in Table 1. The variability in the entry time is mostly due to kernel I-cache misses (1 to 19) and occasional kernel I-TLB misses (1 to 3). From these results we estimate 9.8 to 29.8μsec to process 3 RM cell events. At full line rate of 155 Mbps, with the minimum interrupt period set to one RM period (32 cell times) this overhead will be experienced on average every 86μsec, yielding a host overhead of 11% to 35% for full line rate duplex ABR. Since it is unlikely that the caches will be completely cold after 60μsec, we do not expect to sustain the worst case overhead. In typical use, we expect much less overhead for the following reasons:

- Host ABR processing overhead is proportional to the ABR traffic bandwidth. Such overhead occurs only when transmitting or receiving ABR traffic. All other traffic types, i.e., CBR/VBR/UBR, do not require a host's involvement for flow control. The host overhead is significantly smaller in the presence of other types of traffic and/or when the host is not simultaneously sending and receiving ABR traffic at full line rate. In fact, a 100MHz PC with the current (Winsock) TCP/IP stack is not capable of sustaining data transfer anywhere near 155Mbps.

- When an end system is sending and/or receiving at high rates, the network driver is involved a lot in transmitting and receiving data frames. The driver can also check the HRQ after submitting or receiving a frame to/from the network interface card so that no ABR interrupts need to be generated when the host is performing driver tasks. This can reduce the ABR interrupt overhead and also improve the responsiveness of software ABR flow control.

- Since the processing of ABR requests is not time critical, the host can batch ABR requests to amortize overhead by increasing the minimum interrupt period. A longer interrupt period is equivalent to making the network loop longer.

In any case, the rapid roll out of fast PCs, e.g. 200MHz Pentium Pros, will lead to reduced relative overhead. For QFC, larger switch buffer capacity per VC will also reduce the frequency at which HRQ processing overhead occurs.

To some degree there exists a contradiction between low overhead communication and host-assisted ABR processing. However, host-assisted ABR processing is an indiscriminate drag on all host activity and adds no more overhead on average

[4]Brad Chen assisted greatly, using methodology described in [4].

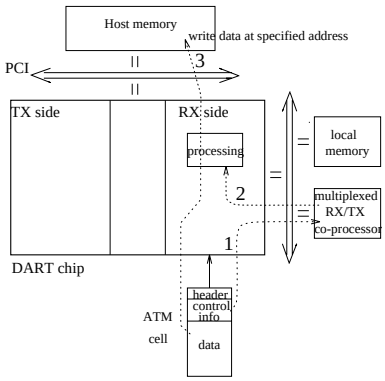

Figure 7: Remote write

for small messages than it does for large messages. Thus eliminating the per message OS overhead for small messages is still important to make them cost effective. In addition, in many cases VBR will be a better choice for low latency, small message traffic than the ATM Forum's ABR scheme.

For more information on DART's traffic management implementation see [14].

4 Message Processing

Message processing is low level processing of frames (i.e. messages) between retrieving a frame from the host and injecting it in the network (TX), and between receiving a frame from the network and storing it to the host (RX). On the RX side, message processing provides a way to examine incoming frames before depositing them in host memory. Possible uses include sender-based protocols, remote queues [2], and remote memory operations for extremely low latency communication (remote reads and writes), host-less filtering of unwanted messages (such as arise with ATM Forum style LAN emulation), and fast/cheap demultiplexing. On the TX side, message processing provides a very low overhead way to inject messages with the appropriate format for message processing at the destination and also provides the mechanism for host-less remote reply operations.

To support message processing, DART includes means for inserting, extracting, and examining "control information" to/from the beginning of messages. This support facilitates vendor and customer experimentation with new developments in communication protocols. DART includes a memory mapped interface to a message co-processor on

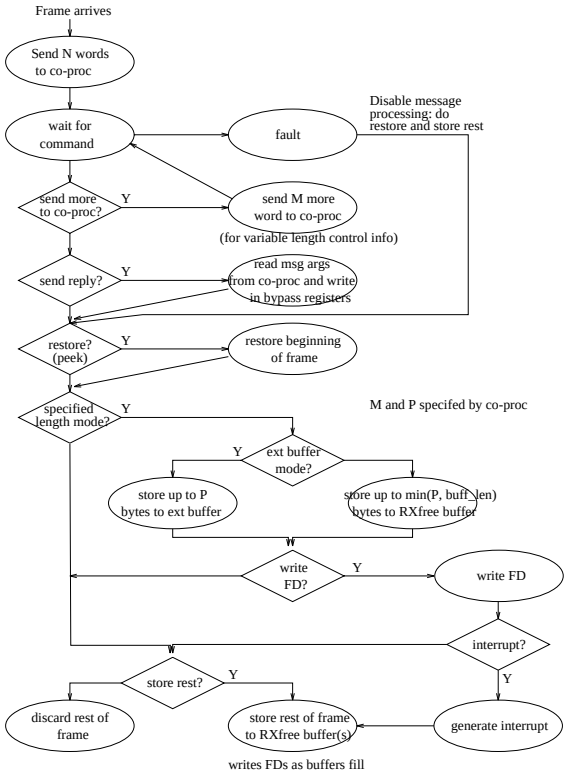

Figure 8: RX side message processing

the DART local memory bus. On receiving a message on a specially marked VC, DART writes a specified part of the beginning of the message to an external co-processor and reads and interprets a simple command returned by the co-processor (see Fig. 7). On the TX side, DART also has a small amount of support whereby DART sends a specified part of the beginning of a message to an external co-processor and reads and interprets a simple command returned by the co-processor.

As an experimental feature, message processing could only impose a minor burden on the rest of DART. One constraint arising was single mastership on the local bus. This precluded a direct connection to a microprocessor to realize the message co-processor and led to the memory mapped interface. However, such an interface allows simple co-processor implementations, such as a FPGA, that can achieve quite elaborate functionality.

Figure 8 shows a flowchart of DART's RX side message processing operation. The diamonds denote commands, which are orthogonal. Table 2 lists a subset of these commands and some example uses. Specified length mode stores up to specified

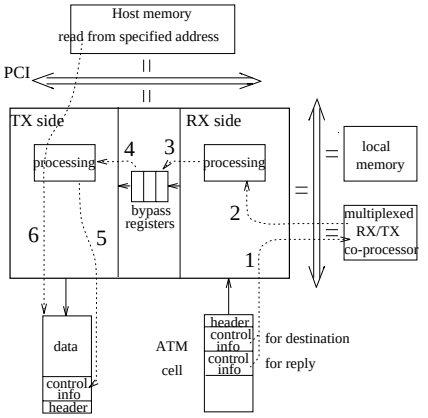

Figure 9: Remote read

amount of data starting at either an buffer address given by the co-processor (the "external buffer") or a buffer address obtained from a RXfree. This second option permits part of frame (e.g. the TCP/IP header) to be stored to one buffer and the remainder (e.g. the data) to following buffers obtained from RXfree.

DART's TX side message processing is simpler. DART sends a fixed number of bytes from the FD (if short mode frame) or first BD (if long mode frame) to the message co-processor so it can determine the operation to perform. The read_coproc command reads a specified amount of data from a given location and prepends it to the front of the out-going frame. For small messages up to 8 bytes in size, it is possible to store the data directly in a short mode FD and have the message co-processor form the outgoing frame using the read_coproc command. With TXrequest in local memory, this option provides very low latency since only a single PCI transaction is required to transfer the FD including data (sourced from the floating point registers).

For implementation simplicity all TX and RX message processing applies only to first cell in a frame.

As demonstrated by Active Message [13] implementations, a host processor can certainly perform all the message processing functions. However, the per-message interrupts of Active messages preclude high bandwidth for small messages. DART's message processing support minimizes host overhead and provides sustained high bandwidth in addition to low latency.

One of most innovative features in DART is special support for request-reply, e.g. remote read. Request-reply is intrinsically difficult to support since the received request must not only be decoded like a remote write, but a send must also be initiated for the reply. DART provides a special path to enable simple request-replies, like remote read, to bypass the host. Referring to Fig. 9, initially DART handles a remote read the same as a remote write: the RX side writes a specified amount of the frame data to the message co-processor and then reads a command. If the command is "send reply", the RX side passes arguments — essentially an FD — to the TX side via the bypass registers and then the TX side takes over. The TX side polls these registers just like a TXrequest queue and upon finding an entry invokes TX side message processing to generate the reply message: the read_coproc command prepends the reply control info to the reply data sent from a host memory location by DMA.

Most of the afore-mentioned uses of message processing are obvious. fetch&add can be implemented using a register maintained by the message co-processor. Remote queues [2], with data stored directly in a named queue, can be implemented using the co-processor to maintain queue pointers: the message control info identifies the queue and the co-processor generates the address at which DART stores the message data. This is a simple form of a virtual ring queue (c.f. Section 2.3.1).

5 Results

Simulation results from the full design Verilog RTL model for two endstations (including PCI bus) connected back to back without a switch show 11μsec application-to-application memory latency for a single cell message. Remote writes and reads of up to 32 bytes take 11μsec and 22μsec respectively. The RX side can process all message sizes at full link bandwidth (155Mbps). There is 1.1μsec startup time per message chain on the TX side (above the cell slot time of 2.7μsec) and thus a stream of single cell messages using short frame format attains 70% of maximum bandwidth.

Using first generation ATM NIC hardware (Fore 100 series NIC) and host software, [7] reported 25μsec for a 32 byte remote write and 45μsec for a 32 byte remote read. These are warm-cache numbers involving about 17μsec of host process-

Command	Example use
deposit specified data size at specified address	remote write of any 4 byte multiple
deposit specified data size in buffer	demultiplex header and data to separate buffers
send more control info to co-processor	variable length control info
interrupt after storing	Active messages
send reply	remote read, fetch&add
abort frame	filtering

Table 2: Example Message Processing Commands

ing for the first cell in a remote memory operation and thus are not sustainable at full bandwidth. Using second generation ATM NIC hardware (Fore 200 series NIC) and i960 firmware, [12] indicated 22μsec for application-to-application latency.[5] Extrapolating from the numbers in [12] suggests about 44μsec for a 32 byte remote read. For two cell messages the one way latency jumps by a discontinuous additional 27μsec [12]. Finally, a stream of single cell messages attains about 30% of maximum bandwidth.

6 Conclusion

DART is a full featured, commercial ATM NIC chip with flow control and supporting 11μsec end-to-end latency at high bandwidth. With a single switch we expect end-to-end latency of 16 to 35μsec, depending on the switch and physical medium. This performance enables a PC/workstation with a single DART interface to support both traditional LAN applications and emerging high bandwidth and low latency applications, such as cluster-based computing. Message processing support provides a kernel for a variety of low overhead communication mechanisms.

References

[1] Blumrich et al, "Two virtual Memory Mapped Network Interface Designs", Hot Interconnects 94

[2] Brewer et al, "Remote Queues: Exposing Messages Queues for Optimization and Atomicity", SPAA95

[3] Buzzard et al, "Hamlyn: A High-Performance Network Interface with Sender-based Memory-management", Hot Interconnects 95

[4] Chen, B. et al, "The Measured Performance of Personal Computer Operating Systems", ACM Transactions on Computer Systems, Volume 14, Number 1, February 1996.

[5] Druschel, P. and Peterson, P. and Davies, B., "Experiences with a High-Speed Network Adaptor: A Software Perspective", SIGCOMM94

[6] Fore Systems Technical Assistance (March 96) and Data Communications, March 95

[7] R. Osborne, "A Hybrid Deposit Model for Low Overhead Communication in High Speed LANs", 4th IFIP Int'l Workshop on Protocols for High Speed Networks, Aug. 1994

[8] "Quantum Flow Control", Version 2.0, Gaddis and Kelt, Eds., July 1995. Available via http://www.qfc.org

[9] Swanson, M. and Stoller, L., "Low Latency Workstation Cluster Communications Using Sender-based Protocols", University of Utah, June 1994

[10] Thekkath, C. and Levy, H. and Lazowska, E., "Separating Data and Control Transfer in Distributed Operating Systems", ASPLOS VI, Oct. 1994

[11] Traffic Management 4.0, ATM Forum Document af-tm-0056, April 1996 (formerly TM95-0013R10, February 1996)

[12] von Eicken et al, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", SOSP95

[13] von Eicken et al, "Active Messages: A Mechanism for Integrated Communication and Computation", Int'l Symposium on Computer Architecture, May 1992

[14] Zheng et al, "Implementation of Flexible ABR Flow Control in ATM Networks", TR96-08, MERL/Mitsubishi Electric ITA, April 1996

Acknowledgements

The authors thank Dr. Masao Nakaya of Mitsubishi Electric and Dr. Tohei Nitta of Mitsubishi Electric Information Technology Center America (ITA) for making the DART project possible. The authors also thank Dr. Hugh Lauer and Dr. Abhijit Ghosh of ITA for their contribution in managing the project.

[5] Half the round trip number in [12] and subtracting 10.5μsec (an overestimate using the latency for SONET [6]) to eliminate switch latency. The DART Verilog roughly simulates a TAXI interface like the Fore NIC uses. Using Mitsubishi's SONET TC chip on each link end would add an average of 6.8μsec latency over TAXI for each link traversal.