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

DART: A LAN Interface for Low Overhead Communication

Randy Osborne

TR94-09v2 December 1994

Abstract

This article presents a low level protocol and network interface architecture for low overhead communication in a distributed memory computing environment — such as workstations and PCs connected via a high speed LAN. We use both sender information and destination information to demultiplex messages directly to where they are needed. The network interface filters incoming messages, separating data delivery from synchronization so as to enable the optimization of simple data delivery while leaving more difficult synchronization to the host processor. To perform this filtering the interface has a small set of simple operations and a small amount of state. We have designed an interface architecture called DART which specializes these ideas to ATM networks. We have built an in-kernel software implementation of this interface with stock workstation and ATM interface cards. This implementation currently achieves a best case application to application latency of 24.5 usec. With a hardware version of DART, we expect to achieve latencies of under 10 usec for a 155Mbps ATM LAN and under 3 usec for a 622Mbps ATM LAN in the workstation LAN environment.

1. First printing, short abtract, May 19, 1994
2. Second printing, final version, July 31, 1994

1 Introduction

This article presents a low level protocol and network interface architecture for low overhead communication in a distributed memory computing environment — such as workstations and PCs connected via a high speed LAN (or perhaps computing nodes in a tightly coupled multiprocessor). By low overhead communication we mean both low latency and low impact on the host processor. We regard high bandwidth in this environment as a solved problem (see e.g. [Dav93, BP93, TS93]). Our objective is a low cost interface to support applications in parallel, distributed, and real-time computing. Low latency is essential for parallel computing. In distributed computing, low latency can help improve the performance of remote procedure call (RPC) based applications and increases the flexibility in structuring a system. Low latency can also be useful for real-time computing, but low impact is far more important to insulate real-time tasks on the host processor from unrelated asynchronous communication events.

There are two key ideas in this article. The first is to use both sender information and destination information to demultiplex messages directly to where they are needed without intermediate copying. This idea exploits the sender's knowledge, shifting more of the burden to the sender, in order to simplify the message processing at the destination. In effect, the sender uses its knowledge to pre-demultiplex the messages. [Se93] termed a sender-based subset of this idea the "deposit" model. In our case, the deposit action (address and interrupt generation) is a function of *both* sender and destination information, so we call our scheme a "hybrid deposit" model. Our variation also supports protected, user level communication. We present a low level hybrid deposit protocol and show how it has flexibility for a wide range of application requirements.

The second key idea is to use the network interface to filter incoming messages. Completely isolating the host processor from all communication events by using a second processor as in the Intel Paragon is expensive and having the host processor handle all communication as in the Thinking Machines CM-5 (at user level and with Active Messages [von92]) impacts the host processor performance significantly. We take an intermediate position and use the interface to separate events by their need for the host processor. Events, such as data delivery, that do not require the host processor are handled directly e.g. by depositing data directly into user memory. Events — chiefly synchronization — that do require the host processor are divided into immediate actions that require service immediately and delayable actions that are accumulated and processed when convenient for the host processor (thereby turning them into synchronous events). This separation arises from two observations. First, the interrupt handling cost for mainstream microprocessors is likely to be significant for the foreseeable future. Second, even if the interrupt handling cost was small, it is not appropriate for the host processor, sitting atop an expensive memory hierarchy, to be involved in all messages. Our idea then, is for the interface to filter incoming messages and direct them to the appropriate level in the hierarchy. Since a design constraint is to stick to stock processor architectures as much as possible, we approximate the memory hierarchy by directly depositing in main memory and interrupting the host processor to deliver data to the top of the hierarchy. Other interface architectures also do this, but either just do simple DMA or are expensive like the high speed MAGIC interface in FLASH [Ke94b]. We are aiming for a minimalist approach to

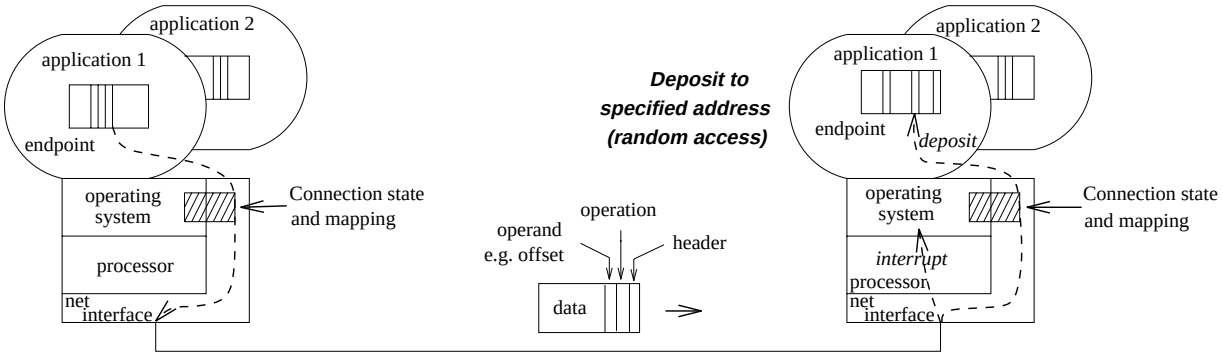

Figure 1: Hybrid deposit operational model

keep the cost low. Thus we endow the interface with a small set of simple operations and a small amount of state.

2 The Hybrid Deposit Model

Figure 1 shows the main idea of the interface: messages originating in a source endpoint can bypass the conventional operating system and host processor route to the network and either be delivered directly to any location in a destination endpoint or conditionally delivered to the destination operating system. The destination address and interrupt generation are a function of both sender and destination state.

Messages contain an ID describing the destination endpoint, control information consisting of an operation (or a pointer to the operation at the destination) and some operands, and data. Each operand is an address, immediate data, or the name of some destination state. Addresses are encoded as an offset from the destination endpoint. The destination end of each communication "connection" has some state, contained in specially addressable locations called "address registers", which message operands can name. Thus message actions are a function of an operation specified by the sender, operands representing sender state, and the contents of the address registers.

The allowable operations are composed of primitive actions that can be implemented simply, in one message time, without host processor or operating system intervention. The host processor can process more complex actions via a software exception. We liken this to RISC philosophy of processor architecture applied to communication.

Figure 2 shows an example set of primitive operations. The few primitive operations in this example set allow a rich set of powerful and flexible compound operations e.g. store indirect with postincrement, priority queueing, and various atomic operations such as fetch-and-increment, compare-and-swap, and barrier synchronization. More complex operations can be achieved using multiple compound messages.

For further information on the hybrid deposit model see [Osb94].

Address generation	effaddr= *operand*	(direct addressing)
	effaddr= $< addreg_i >$	(indirect addressing)
	effaddr= $< addreg_i > + operand$	(indexed addressing)
Register operations	$addreg_i \leftarrow operand$	
	$addreg_i \leftarrow$ **unary-op** $\{ < addreg_j >$ or *operand* $\}$	
	$addreg_i \leftarrow < addreg_i >$ **binary-op** $\{ < addreg_j >$ or *operand* $\}$	
Conditional operations	if ($< addreg_i >$ **compare-op** *operand*) then generate interrupt at end	
	if ($< addreg_i >$ **compare-op** $< addreg_j >$) then generate interrupt at end	

Figure 2: Primitive operations

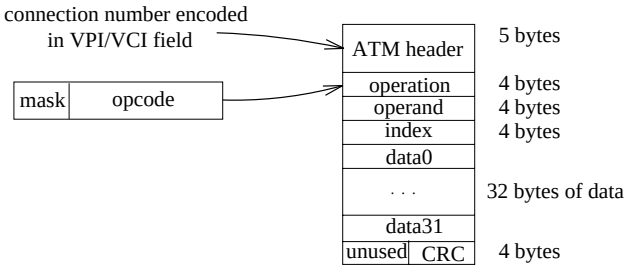

Figure 3: Format of 53 byte ATM cell for hybrid deposit model

3 DART

We have designed an interface architecture called DART which specializes our ideas to ATM networks. The 48 byte payload of an ATM cell contains 32 bytes of data — a size selected to match memory and cache (sub)block sizes — and 16 bytes of control which includes the operation, an immediate source operand (offset or data), and an index field as shown in Figure 3. Destination operands are specified via up to three separate register indices encoded in the index field. Reads and writes occur in 32 byte blocks. The mask field can be used to deselect the reading or writing of 4 byte words within such a block. There is also a multiple cell message format for block transfer. In this format, the first cell is a "control" cell in the above format and the following cells are standard AAL5 cells with 48 bytes of payload for data.

Figure 4 shows a block diagram of DART. The main features are protected, user level interaction and virtual mapping of endpoint regions. The send and receive sides multiplex the Connection and Operation Logic shown in Figure 5 (latches and control signals omitted for clarity). The Connection Table indirects to an Endpoint Table, allowing multiple connections to share the same endpoint. The Endpoint Table contains base and bound virtual addresses pairs defining the contiguous region of each endpoint. Once an endpoint virtual address is formed in any of a variety of ways, it is bounds checked, and then mapped to a physical address by the TLB. Consequently, endpoints need not be pinned in main memory as is common in other interface designs. Each connection has a "window" of address registers demarcated by the base and bounds entry in the Connection Table. This scheme allows a variable number of address registers per connection and allows the overlapping and nesting of register windows

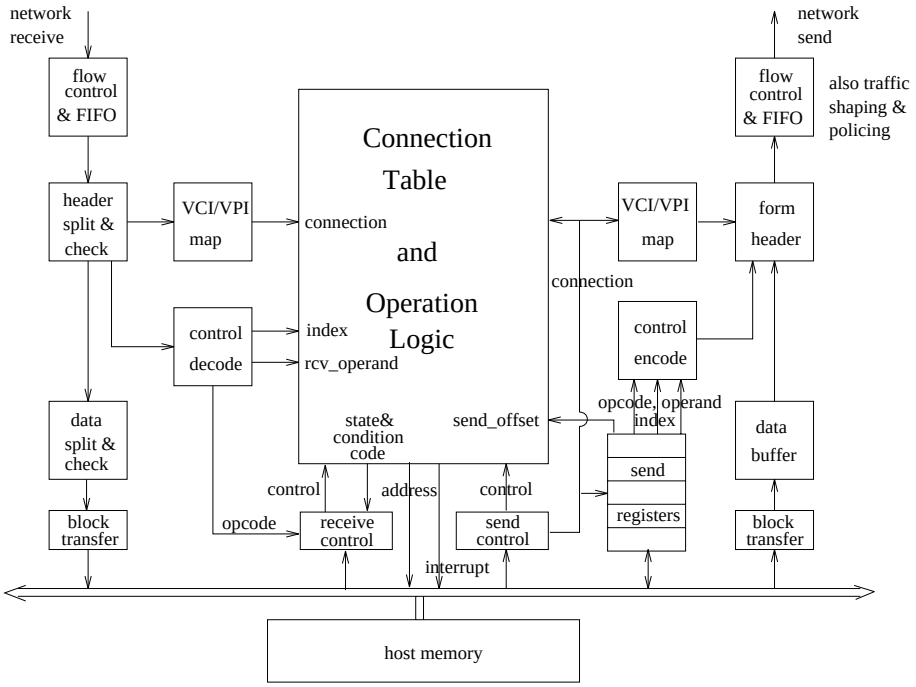

Figure 4: DART block diagram

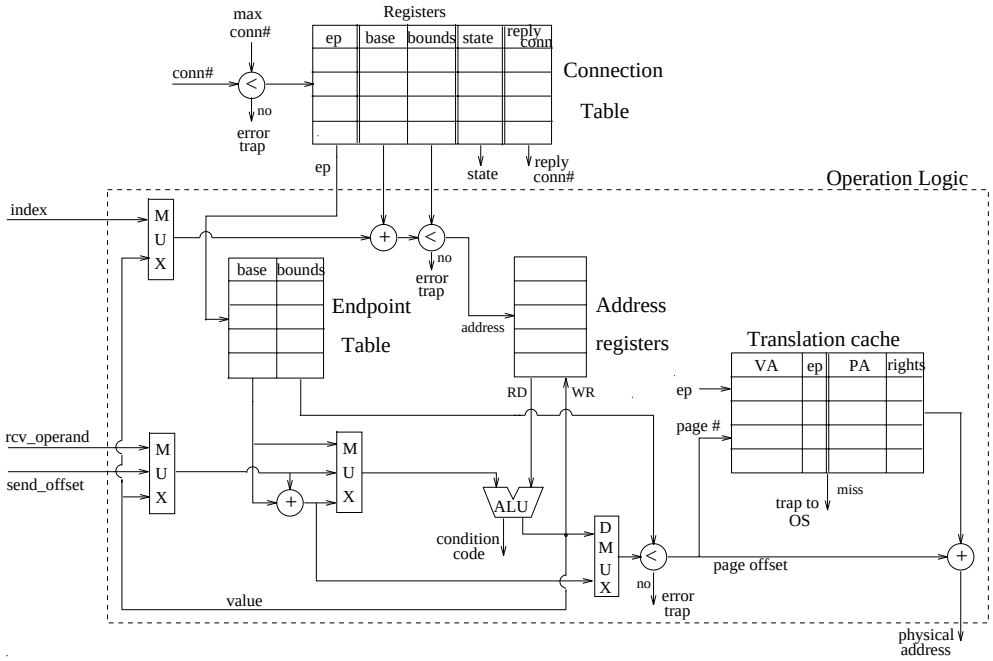

Figure 5: Connection Table and Operation Logic

to provide flexibility in sharing and protection between connections.

Each connection has a set of memory mapped Send Registers. To send a message from a given endpoint, a user writes control information, the message's offset from the endpoint base, and the data size into the Send Registers for an appropriate connection for that endpoint. The control information consists of an opcode, source operand, and address register indices for depositing the message at the destination.

DART can execute up to four primitive operations per message received (one address generation, two register operations, and one conditional). We clock the Operation Logic through multiple primitive operations per message, feeding back immediate values as necessary via the "value" path shown in Figure 5. A main opcode controls the selection and ordering of the primitive operations. Example opcodes are read, read multiple (32 byte blocks), write, write multiple, and software exception (which causes an interrupt to the host processor). Remote reads are handled without the interrupting the host processor. Exceptions arising due to error traps, TLB misses, and unimplemented operations cause an interrupt to the host processor.

A variant of DART interprets the operation field as an "instruction" pointer to an opcode and operand at the destination (see [Osb94] for details). Many implementation details require further evaluation before DART implementation can begin. We are targeting the PCI bus and plan to use a cheap i960 embedded processor to implement the Operation Logic.

We have built a software implementation of the hybrid deposit model to simulate the interface described above. This is an in-kernel implementation under Mach 3.0 on a DECStation 5000/240 with a 140Mbps Fore Systems TCA-100 ATM interface card. The best case latency for a 32 byte application to application data transfer is 24.5μsec on DECStation 5000s connected via a switchless ATM network and 30μsec on DECStations connected via a single Fore Systems ASX-100 ATM switch. This is about 10 times faster than the fastest conventional approach (using Fore System's AAL3/4 implementation) on the same hardware[Ke94a]. With our DART interface, we expect to achieve latencies of under 10μsec for a 155Mbps ATM LAN and under 3μsec for a 622Mbps ATM LAN in the workstation LAN environment (for one transit of a fast ATM switch). We plan to use this software implementation to evaluate the hybrid deposit model and DART for higher level protocols and applications.

4 Related Work

The per cell processing architecture of DART is similar in principle to the message-driven processor (MDP) [De87]. However, we use DART in a filtering role for depositing messages, rather than for direct computation (i.e. it is not the main processor) and we provide full protected, multiuser communication. SHRIMP [Be94] and Hamlyn [Wil92] adopt a similar emphasis on increasing the participation of the sender to minimize the required functionality at the destination. However both differ from DART in some major ways: they both support only direct addressing and both pin endpoint pages. In addition, Hamlyn supports only one delayed action queue per node whereas there can be any number in DART. A simpler version of SHRIMP has a limited form of hybrid addressing without indirection [DLM94]. MINI only

supports direct addressing [MBH94]. Like DART, Axon [SP90] has self describing packets for direct deposit at the destination. However, Axon lacks hardware support for flexible sender and destination-based addressing, hybrid interrupt control, and register operations.

Acknowledgments

Development of the hybrid deposit model benefited from discussions with Peter Steenkiste and Thomas Gross at CMU.

References

[Be94] M. Blumrich and et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Intl Symposium on Computer Architecture*, April 1994.

[BP93] D. Banks and M. Prudence. A High-Performance Network Architecture for a PA-RISC Workstation. *Journal of Selected Areas in Communications*, pages 191–202, February 1993.

[Dav93] B. Davie. The Architecture and Implementation of a High-Speed Host Interface. *Journal of Selected Areas in Communications*, pages 228–239, February 1993.

[De87] W. Dally and et al. Architecture of a Message-Driven Processor. In *Intl Symposium on Computer Architecture*, 1987.

[DLM94] C. Dubnicki, K. Li, and M. Mesarina. Network Interface Support for User-Level Buffer Management. In *Parallel Computer Routing and Comm. Workshop, Univ. of Washington*, May 1994.

[Ke94a] P. Keleher and et al. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of Winter Usenix Conf.*, January 1994.

[Ke94b] J. Kuskin and et al. The Stanford FLASH Multiprocessor. In *Intl Symposium on Computer Architecture*, April 1994.

[MBH94] R. Minnich, D. Burns, and F. Hady. A 1.2Gbit/sec, 1 microsend latency ATM Interface. In *Hot Interconnects II*, August 1994.

[Osb94] R. Osborne. A Hybrid Deposit Model for Low Overhead Communication in High Speed LANs. 4th IFIP International Workshop on Protocols for High Speed Networks., August 1994.

[Se93] J. Subhlok and et al. Programming Task and Data Parallelism on a Multicomputer. In *Proc. of ACM Sympos. on Principles and Practice of Parallel Programming*, pages 13–22, May 1993.

[SP90] J. Sterbenz and G. Parulka. Axon: A High Speed Communication Architecture for Distributed Applications. In *Proceedings of IEEE INFOCOM*, 1990.

[TS93] C. Traw and J. Smith. Hardware/Software Organization of a High-Performance ATM Host Interface. *Journal of Selected Areas in Communications*, pages 240–253, February 1993.

[von92] von Eicken et al. Active Messages: A Mechanism for Integrated Communication and Computation. In *Intl Symposium on Computer Architecture*, pages 256–266, May 1992.

[Wil92] J. Wilkes. Hamlyn: An Interface for Sender-based Communication. Technical Report HPL-OSR-92-13, HP Labs, November 1992.