RC 20445 5/30/96
Computer Science

Research Report

Using Commodity Networks for Distributed Computing Research

Ronald Mraz, Douglas Freimuth, Edward Nowicki and Gabriel Silberman

IBM T. J. Watson Research Center
Yorktown Heights, New York

Research Division
Yorktown Heights, New York • San Jose, California • Zurich, Switzerland

Using Commodity Networks for Distributed Computing Research [1]

Ronald Mraz, Douglas Freimuth, Edward Nowicki and Gabriel Silberman

Wideband Processor and I/O Architectures

IBM T. J. Watson Research Center

Yorktown Heights, New York

Ph. +1-914-945-2109, Fax -4425

{mraz,freimuth,nowicki,gabby}@watson.ibm.com

Abstract

We describe a zero-copy network programming interface that is suitable for parallel processing, local area networks and real-time multimedia applications. This interface is suitable for directly programming applications or as an intermediate layer for existing distributed computing libraries. Using this approach, buffers in host memory are directly managed by the network hardware interface, thus eliminating the need for copying data from network interface buffers to host memory. A distributed system, made up of commodity workstations interconnected by dedicated ATM hardware, functions as a custom packet switched interconnection network typically used in high performance parallel machines.

1 Introduction

Local area networks have long been considered as candidates for parallel processing clusters [1] [2] [3]. Nevertheless, they are hindered by the latency penalties imposed by current global communication protocols. For example, protocols such as TCP/IP and UDP [4] add latency by requiring 16-bit checksum calculations and packetization that typically involves segmentation and re-assembly of data. Furthermore, these protocols require task switching between user and system domains to prepare the data for transmission and reception. Each operation buffers and copies the data at each transition point [5] impacting the overall system performance. An emerging protocol standard for communications, Asynchronous Transfer Mode (ATM) [6], allows for a streamlined software path. The efficiency of this approach is due to the migration to the hardware "link" layer of many of the tasks previously done in software.

[1] The opinions expressed herein are those of the authors, and do not necessarily reflect the opinions of International Business Machines Corporation.

The result is a commodity interconnection network which has the potential to rival custom networks in latency and bandwidth. To realize this potential, the overhead of the software, as seen by the user's application, has to be kept low. This is done by providing a communication layer that implements a "zero-copy" transfer from user memory space to network media, e.g., fiber or copper wire. We have termed this low level, high performance interface, *Zero-copy ATM (ZATM)*. The uniqueness of ZATM is not limited to low software latency, but also includes a unified *Application Programming Interface (API)* to service applications in such diverse domains as parallel processing, local area networks and real-time multimedia. Not only are the API calls common to the above areas, but much of the device driver handling for these calls is also common.

The next section describes the goals and the design philosophy behind ZATM, including the functional partition between software and hardware. Section 3 describes the interface between the hardware and the device driver. Section 4 describes the ioctl ZATM call interface. Section 5 provides preliminary performance figures. Section 6 provides summary, conclusions and future work.

2 Software Architecture Considerations

The objective of this work is to provide a consistent zero-copy interface for parallel processing libraries such as PVM or MPI, local area network interfaces such as UDP and TCP/IP, and emerging real-time multimedia applications. Typically, these functions would use different low level interfaces, or build upon higher level interfaces such as TCP/IP, thus sacrificing performance in exchange for a consistent user interface.

Each domain of computing that we wish to support requires a separate protocol specification, including features not required in the other domains. As an example, distributed memory parallel processing requires both asynchronous and blocked (synchronized) send and receive operations directly from user space. On the other hand, LAN interfaces require the ability to separate the header information from the data without introducing any overhead and interrupt currently executing user processes for service from system space. Further, real-time protocols require the ability to guarantee that a message is either sent or received within specific time bounds. In crafting an interface to provide these functions, a non-intuitive result was found, namely, that these requirements cooperate rather than compete for implementation details. This provides a very streamlined set of code that have few special cases and is consistent across each user domain.

This cooperation among domains is best captured by adding a time parameter for each blocking send and receive call. If a message is not sent or received within the time specified by the user, the function call times out and its return code reports this fact to the caller. This feature not only satisfies the real-time message requirements, but it obviates the need for separate code paths for asynchronous and blocked operations. In this scheme, asynchronous sends and receives simply become blocked operations that have a timeout of zero time units.

Additionally, for parallel processing, we can use this parameter to detect application deadlock conditions. This is done by setting the timeout to something greater than the maximum time a process should wait for a particular message. Depending on the desired robustness of the application, recovery can be attempted after the timeout.

LAN interfaces that use system level messages for configuration and set-up require timely responses. Critical timings can now be incorporated at the network interface. An example would be in distributed time synchronization, where a round trip message between nodes that are attempting to coordinate their clocks needs to be completed within a certain time period. This check is currently being done at the application interface.

In the following section we describe the hardware interface used by ZATM, and how it fulfills its function.

3 Hardware Description

The particular hardware interface used by ZATM, a prototype built at IBM Research called *Cheetah*, consists of two highly integrated chips, complemented by 512K bytes of memory (for tables) and an interface to the network media (fiber). One of the chips performs the *Segmentation and Reassembly (SAR)* function, including handling of the ATM AAL5 DMA protocol and interface to the processor's PCI bus. The second chip translates signals between the digital domain and the levels required to drive the ATM network interface. Notice that there is no microprocessor in Cheetah, thus making it simple, fast, and with potential to bandwidth improvements, not limited by a processor's speed. ZATM makes full use of the special features in the integrated PCI/ATM SAR chip. Notably, the full hardware state machine implementation of ATM AAL5 DMA protocol.

Cheetah is a fully integrated PCI to ATM interface that provides full-duplex Segmentation and Reassembly (SAR) at 155 Mbps "wire-speed" (310 Mbps aggregate speed). Data transfer and control information is done through Direct Memory Access (DMA) transfers to/from the host computer's main memory. This is done asynchronously for both sending and receiving data.

To send a buffer of arbitrary size, the device driver builds a *Transmit Buffer Descriptor (TBD)* and places this information in a predetermined pinned area (wired) in host memory. The driver then updates the head pointer for the queue and writes this value to a register in the ATM interface chip. The interface chip automatically reads and parses the TBD to obtain the ATM *Virtual Channel (VC)* number, the length of the data for the message, and the address of the data in host memory. The interface chip then DMAs the message, in 48 byte cells for standard ATM data payloads, and transfers the data to the physical fiber interconnect. A checksum is calculated "on the fly" and appended to the last ATM cell for checking on the receiver.

At the point that all data has been successfully transferred to the physical interface, an entry is placed in the *Transmit Status Queue (TSQ)* corresponding to the parameters provided in the original TBD. In this way, the device driver knows exactly when a message has been sent. The ZATM interface (described in the next section) makes use of this information to provide deadline guarantees for message transmission by including *time* as a measure of success in any operation.

To receive a message, the interface chip has a pool of buffers that have been pre-allocated with DMA priviledges. Each 48 byte cell payload is encapsulated with an additional 5 bytes of header information to identify the virtual channel (and other infromation) associated with the payload. The interface chip parses this information and re-assembles the original message in one or more of the predetermined buffers. Furthermore, the chip provides a checksum "on the fly" as the data is DMAed to host memory.

Once the message is reassembled in host memory, a set of one or more *Receive Status Descriptors (RSD)* are present in the *Receive Status Queue (RSQ)* in host memory. The driver can then parse the RSD to determine the location of the data, the VC number, success or failure of the checksum, etc. Given that the transfer can be contained in a single 4096 byte buffer, the application can use this data directly since it has been completely reassembled.

To transfer the message from system to user space and vice-versa, the IBM AIX operating system [7] provides a virtual memory architecture that is highly optimized for such transfers. In this model, user and system pages of all processes map into a single monolithic paging area. This allows kernel copyin() and copyout() functions to simply update virtual memory mappings rather than copy data. In this way, we have transferred data from the ATM interface chip to host memory, in either user or system space, *without* any copying of bytes for both sending and receiving of data.

4 ZATM Programming Interface

Traditional Application Programming Interfaces (APIs) for parallel processing provide two flavors of transfers, blocked and asynchronous transfers. A "blocked" transmission is an operation where the application is blocked from executing until the device driver finishes transferring the data to or from the physical interface. On the other hand, an "asynchronous" transmission is considered as completed, from the application's point of view, when data has been copied to a buffer in the device driver for subsequent transfer. LAN interfaces (such as UDP, etc.) would first copy the data into a buffer for subsequent transfer.

To provide deadline guarantees for ZATM operations, the notion of time is provided for all operations. An operation must not only perform the intended transfer, it must perform the transfer within a specific amount of time for it to be successful. To determine the success or failure of an operation the device driver can monitor the send status and receive status queues. Therefore, all operations can be assigned a duration within which the operation must complete for a successful transfer to occur.

To perform an asynchronous transfer (*zatm_send or zatm_recv*) the timeout parameter is set to zero. This allows the driver to return immediately without checking for status of the sent message. To perform a blocked send (*zatm_bsend or zatm_brecv*) the device driver simply waits for the matching entry in the TSQ to appear. In all transfers, the device driver maintains a timeout parameter that is set for the maximum interval that the application can wait for the transmission to occur.

This operation is particularly useful in scheduling real-time messaging applications to insure that sending occurs within a predetermined time interval. If the duration is too long, the application (or driver) can take corrective action. This action would involve increasing the utilization of the send queue or defining a separate queue for this virtual channel that results in higher utilization. All of these operations can be set in hardware by the device driver.

This is particularly useful for continuous media applications, such as a video service, that require predictable utilization of the network interface. If this condition were not detected or corrected, the receiving end could experience uncorrectable jitter. Adding the dimension of time simplifies rather than complicates the interface for blocked and asynchronous transfers, while providing the ability to detect exception conditions for real-time operation.

5 Preliminary Results

ZATM has been implemented on a early prototype of an integrated PCI/ATM interface chip (IDT 77201 [8]) and will be fully functional when production hardware is available. The interface described makes full use of all features of the integrated PCI/ATM Segmentation and Reassembly (SAR) chip.

The initial development of the protocol was done in the *Naked Application Programming* environment [9]. The NAP environment provides a means to evaluate and quantify features of ZATM without introducing variance of an operating system. Average latencies observed for 10 minimum size message bursts of 40 bytes (each message contains 40 bytes data, 8 bytes ATM cell trailer) memory to memory is 45 microseconds. The initial transfer has an additional startup overhead of 30 microseconds. This delay is due to the send logic's pipeline interface to the ATM serializer component. Analysis of the hardware in an AIX environment is ongoing.

6 Conclusions and Future Work

The goal we set to ourselves for the design of ZATM, a zero-copy ATM network programming interface, was to streamline the software overhead from an application's perspective. This so called "protocol stack" is a serious impediment to achieving inter-processor communications with latency and bandwidth close to the network's capabilities. Since we were also considering ZATM's applicability to parallel processing, local area networks, multimedia and real-time applications, it was paramount for the interface to be both simple as well as unified as possible.

Partitioning of the tasks between hardware and software, as enabled by the Cheetah adaptor, resulted in a device driver which imposes a very light load on the CPU, while managing receive and send buffers in the CPU's own physical memory space. As seen by the preliminary performance figures, we feel we are on the right direction to attain the original design goals.

As part of our initial suite of applications, we are demonstrating the ZATM interface as part of a system to both capture and playback video on demand at Supercomputing '95. This system captures video at the lecture halls and exhibit floor, stores it in an IBM SP2 system, and makes it available on demand at several kiosks throughout the conference floor, and at a couple of locations off-site. The ZATM/Cheetah combination was installed on PowerPC 604 boxes (model 43P) to perform the transfer via ATM/OC-3 links of both video and audio using the RTP protocol within the video application ported to IBM platforms by Argonne National Labs.

Several other parallel programming environments are planned to be experimentally ported on top of ZATM, including IBM's High Performance Fortran (HPF), Earth-Manna from McGill University [10], MPI [11], as well as general TCP/IP for LAN communications. Other, more innovative environments, will also be evaluated within the ZATM framework.

Clearly, interfacing to ATM through a PCI bus has its limitations, when considering parallel systems with tight coupling of the processors. It is thus our intention to investigate the next logical step beyond ZATM/Cheetah, which is to interface ATM, or other low-latency, high-bandwidth network, directly into the processor's memory bus.

7 Acknowledgements

We acknowledge B. Olson and I. Judson of Argonne National Labs, and E. Jedlicka, T. Dirks and R. Demkowicz from IBM, for their work on the SC '95 Demo, W. Kleinfelder of CCS and R. Minnich from SRI on their vision for low-latency ATM, and D. Zumbo and M. Thoennes of IBM Research who provided the hardware support.

References

[1] L. Choi and A. Chein, "Integrating Networks and Memory Hierarchies in a Multicomputer Node Architecture," *IEEE Proc. of the Intl. Conference on Parallel Processing*, p 10-17, October, 1994

[2] J. Heinlein, K. Gharachorloo, S. Dresser and A. Gupta, "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor," *IEEE ASPLOS-IV*, 1994, p 10-17, October, 1994

[3] Arvind, "StarT the Next Generation: Integrating Global Caches and Dataflow Architecture," *MIT Laboratory for Computer Science,* CSG Memo 354, February, 1994

[4] D. Comer and D. Stevens, *Internetworking with TCP/IP, Volume II, Design, Implementation, and Internals*, Prentice-Hall, 1994.

[5] D. Saha, D. Kandlur, T. Barzilai, Z. Shae and M. Willebeek-LeMair, "A Video Conferencing Testbed on ATM: Design, Implementation and Optimizations," *IEEE Intl. Conference on MultiMedia Computing and Systems*, pp. 23-30, May 1995.

[6] The ATM Forum, *ATM User-Network Interface Specification*, Prentice Hall, 1993.

[7] Technical Reference:, "Kernel and Subsystems" *AIX Version 4.1 for RISC System 6000*, 1994

[8] Technical Reference:, "NICSTAR: IDT 77201 User Guide and Programming Manual" *IDT Public Domain Document*, Preliminary release C

[9] Technical Reference:, "NAP: Naked Application User Guide and Reference" *IBM Public Domain Document*, 1994, A NAP tarfile for the IBM 40P is available from the ftpsite: liber.stanford.edu

[10] O.C. Maquelin, H.H. Hum and G.R. Gao, "Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors," *Proceedings EUROPAR95*, 1995.

[11] W. Gropp, E. Lusk and A. Skjellum, *Using MPI*, MIT Press, 1994.