An Intel IXP1200-based Network Interface

Kenneth Mackenzie, Weidong Shi, Austen McDonald and Ivan Ganev
Center for Experimental Research in Computer Systems (CERCS)
Georgia Institute of Technology
Atlanta, Georgia 30332–0280
{ kenmac, shiw, austen, ganev }@cc.gatech.edu

***Abstract*— We describe and evaluate a quad 100T ethernet network interface built using an Intel IXP1200 network processor on a commonly available Radisys ENP2505 PCI board. The network interface exports a raw ethernet interface either to the host kernel or to user level for cluster computing applications. We describe the firmware architecture and internal design decisions, then evaluate the resulting network interface against 100T and gigabit network interfaces using CLF, a lightweight reliable datagram layer.**

We find that the new network interface provides full bandwidth (196-373Mb/S, depending on packet size) for four 100T ports and a host-to-host, minimum-size-message latency of 28.3uS (17.5uS of which is in our ethernet switch). We estimate the design uses about 40% of the resources of the IXP1200 at 232MHz, leaving ample headroom for application-specific packet processor or for additional/faster ports. As a side note, we observe that it is easy to trunk together multiple 100T links when working with raw ethernet packets in a cluster, unlike working atop IP.

I. Introduction

Network processors are an attractive media for implementing smart network interfaces for cluster computing applications. We have implemented a baseline network interface on an IXP1200-based PCI card as part of a larger project for exploring network interfaces that transform packet data on the fly. In this paper we detail our firmware architecture and design tradeoffs, report measured performance in hardware on our cluster and estimate the headroom left over in the IXP1200 for additional packet processing or for faster links. We conclude by summarizing initial results for transforming kernels, work that we have not yet integrated with the network interface. We observe that IXP1200 appears most useful for kernels that perform a small amount of computation, about two operations per byte transfered.

The IXP1200 [2] forms the core of the network interface hardware (Figure 1). The IXP1200 chip includes six 4-way-multithreaded "microengines" for data movement, a StrongARM for management functions, local SRAM and DRAM and an integral PCI interface with two DMA engines. The Radisys ENP2505 board [7] includes a 232MHz IXP1200, 256MB DRAM, 8MB SRAM, a 100T MAC with four ports and a PCI bridge. The StrongARM core runs Linux and in our design is used only for initialization and debugging.

The rest of the document is organized as follows: Section II presents the architecture and design decisions. Section III presents experimental results. Section IV presents a study of application extensions. Section V presents related work and Section VI concludes.

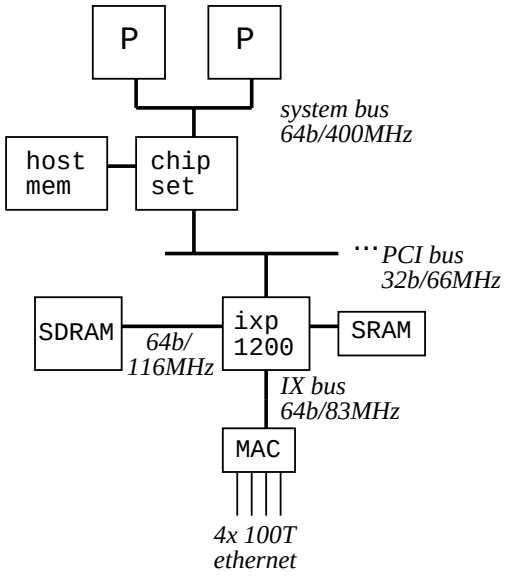

Fig. 1. Hardware architecture: the IXP1200 network processor serves as the bridge between the host and 4x 100T ethernet interfaces using its built-in PCI bus.

II. Architecture and Implementation

Our design uses host software and network processor firmware to implement the network interface. In this section, we describe the hardware and then sketch the key design elements for the software: the shared memory interface between the host and the multithreaded firmware on the IXP1200. We finish by analyzing the amount of IXP1200 resources used and by showing how we intend to implement user extensions to the firmware.

In a nutshell: (1) the host and IXP share two queues in physical memory, one on the host and the other on the IXP, so that all communication across the PCI bus is in the form of writes; (2) the firmware includes features to allow the contiguous sequence of messages from the host to be processed out of order, to reduce shared queue state variable writes across the PCI bus and to combine small messages for better PCI utilization; and (3) the resulting design uses about 40% of the IXP1200's resources, leaving room to add application-specific extensions or more/faster ports.

A. Shared Data

Both the IXP1200 and the host have local DRAM visible to the other. After benchmarking the PCI bus and the IXP1200's DMA capabilities (section III-A), we settled on a design in which the host and IXP share two queues in physical memory, one on the host and the other on the IXP, with state variables arranged so that all communication across the PCI bus is in the form of writes.

The IXP1200 and host communicate through two shared buffers, UP (IXP1200-to-host) and DOWN. Each is a large (1MB), physically contiguous region of memory treated as a circular buffer of variable size messages plus shared HEAD and TAIL indices as state variables. Messages are inserted at the TAIL and removed at the HEAD. We arrange the state variables so that they are written remotely and read locally. I.e. the TAIL on the DOWN buffer is in IXP1200 DRAM: the host writes TAIL when it inserts a message and firmware on the IXP1200 reads TAIL when polling to discover the existence of a new message.

Messages contain a 64-bit header (a message type and a length in bytes) followed by a variable amount of data as payload. The payload here is always an ethernet packet including the 14B ethernet header but without the CRC. Messages are kept contiguous despite the circular buffer by using a message with a reserved WRAP type as a noop to fill the rest of the buffer when the next data message would not fit in the buffer.

After benchmarking the PCI bus with this hardware, we made the following additional design decisions:

- The DOWN buffer is located in IXP1200 DRAM and written via PIOs from the host. Due to a known bug in the particular chipset we have (Intel 860), the bandwidth of host PIOs (198MB/S) is roughly double that of DMA reads (85MB/S) so we stuck with PIOs. The IXP1200 DRAM is not cache coherent to the host – the region is marked uncached with write-combining enabled. The microengines on the IXP1200 have no cache.
- The UP buffer is located in host DRAM and written by DMA initiated by IXP1200 firmware. DMA writes achieve 204MB/S for large packets. Host caches are kept coherent by the host's chipset.
- The IXP1200 uses polling to discover the arrival of messages in the DOWN buffer. Since there are many hardware threads available on the IXP1200, polling has low overhead.
- We have implemented both polling and interrupts for the host. We use polling with our user-mode implementation and we use interrupts (with a tail-polling optimization) with the in-kernel implementation.
- Messages are aligned on 64-byte boundaries in both buffers. Both the host and IXP1200 prefer this alignment: host caches and write-combining buffers use 64B blocks and the IXP1200 DMA engine prefers 64B alignment for transfers. Matching the cache block size and alignment ensures that messages to the UP buffer are transferred efficiently as a sequence of WRITE-INVALIDATE transactions on the PCI bus.

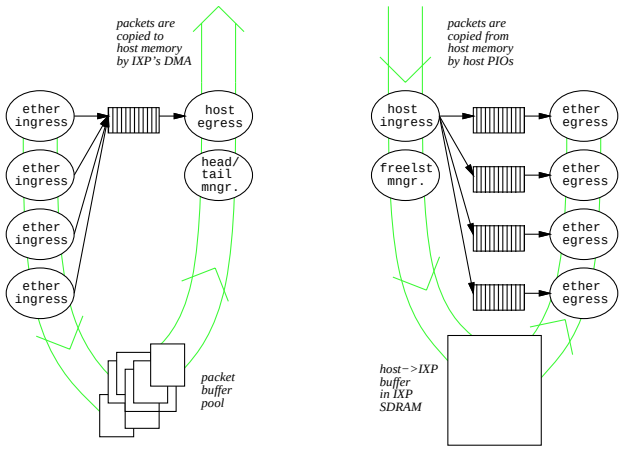

Fig. 2. Firmware architecture. The ingress and egress sections operate independently, each with a thread per ethernet port, plus two threads managing the buffers shared with the host.

The interface to the two shared buffers described above is implemented by the host and by firmware on the IXP1200, described next.

B. IXP Firmware

The 24 threads of the IXP lend themselves naturally to a multithreaded software architecture. We use 12 threads on four microengines in the network interface. The thread assignments are depicted in Figure 2 and correspond more or less to one thread per I/O device per direction (ethernet port or host). The threads are coupled by queues kept in SRAM that point to packet buffers kept in DRAM.

We describe the firmware by following the two major data paths: host-to-net and net-to-host. A major design issue on the host-to-net path is permitting the in-order messages in the DOWN buffer to complete out-of-order when sent to different interfaces. The major issue on the net-to-host path is the latency of invoking a DMA operation which initially formed the bottleneck for small packets.

a) Host-to-Net Path: The host-to-net path uses six threads (Figure 2). HOSTINGRESS polls for incoming packets in the DOWN buffer and dispatches them to ethernet ports. Four ETHEREGRESS threads, one per port, manage the ethernet ports. Each ETHEREGRESS thread has an associated queue in SRAM, entries of which point to packets in the DOWN buffer. Finally, an auxiliary "freelist manager" thread keeps track of DOWN buffer entries and updates the DOWN buffer HEAD pointer as needed. An SRAM data structure (the "freelist"), not shown, is shared by the HOSTINGRESS and freelist manager threads to track free DOWN buffer entries.

The microengines perform minimal data movement in the DOWN path because the host plants the packets directly in IXP1200 DRAM. The host uses PIOs to fill entries in the DOWN buffer before invoking the HOSTINGRESS thread by updating the DOWN buffer TAIL index. The HOSTINGRESS

inspects the first few words of the message to check the message type (for the WRAP case) and to check the ethernet source MAC address in order to determine to which port it should forward the message.

The most interesting part of the host-to-net path is the freelist which permits packets to be consumed out-of-order even though the DOWN buffer operates in-order. The freelist is a circular buffer, sized to be at least as large as the sum of the ETHEREGRESS SRAM queues, and operates in a manner analogous to a reorder buffer in a superscalar processor.

- The HOSTINGRESS thread makes a freelist entry for each message that it forwards to a port and a pointer to this freelist entry is included in the SRAM queue entry along with the pointer to the packet in DRAM.
- Each freelist entry contains a HEAD index value (representing what HEAD should be when this message is freed) and a boolean, initially set, indicating whether this message is in use.
- When an ETHEREGRESS thread finishes sending a packet, it clears the in-use boolean in the freelist entry for the message that encapsulated the packet.
- Meanwhile in the background, the freelist manager thread scans the freelist, copying HEAD values to the host-visible HEAD variable when it finds them empty. To reduce the polling implied here, we use an interthread signal so that the freelist manager only wakes up when there's potential work for it to do.

The DOWN buffer entries thus "issue" in HOSTINGRESS in-order, they "execute" in the ETHEREGRESS threads out-of-order and then they "commit" in the freelist manager in-order. We use the freelist mechanism to allow messages to different ports to complete out of order. The mechanism also could be used to cover long-latency operations in an extended version of the the network interface as well, e.g. application-specific operations on messages by spare microengine threads. Its limitation is that all buffers must eventually be freed to avoid deadlocking the DOWN buffer.

b) Net-to-Host Path: The net-to-host path also uses six microengine threads. Four ETHERINGRESS threads are used, one thread per port, in the style recommended by the IXP1200 documentation and Intel's standard macro package. The four ETHERINGRESS threads receive into one queue which is then read by the HOSTEGRESS thread. The HOSTEGRESS thread schedules the DMA engine on the IXP1200 chip to transfer data to the UP buffer on the host. An auxiliary "head/tail manager" thread uses DMA to copy the UP buffer TAIL index (and, incidentally, the DOWN buffer HEAD index) to host DRAM.

The major issue for the net-to-host path is providing acceptable bandwidth for small packets such that the network interface is not a bottleneck. The DMA engine achieves great bandwidth for large packets but is slow to start and so is a challenge for small packets. We use three techniques to deal with DMA: (1) we pipeline DMA requests to overlap invocation time, (2) we adaptively reduce the frequency of HEAD and TAIL updates under load and (3) we combine small messages under load. Techniques (2) and (3) also serve to reduce the frequency of host interrupts.

1) DMA invocation is pipelined to overlap invocation time. Launching a DMA for a minimum-size message takes three roughly equal-latency actions, all accesses to DRAM: (a) microengine thread writes descriptor to DRAM, (b) DMA engine reads descriptor, (c) DMA engine copies data from DRAM to PCI bus. We pipeline (a) with (b)/(c) by using a pair of descriptors in double-buffer fashion, leading to about 1/3 less time per message or 50% more bandwidth. Pipelining all three actions is conceivable because the IXP1200 has two DMA engines. However, the bookkeeping is significantly more complex and we have not yet tried it.
2) We reduce DMA requests for the UP buffer TAIL index and DOWN buffer HEAD index by generating fewer requests when under load. These state variables on the host must be updated by DMA, occupying the DMA engine when it could be moving data. We assign management of the state variables to a separate head/tail manager thread. The HOSTEGRESS and freelist manager threads signal the head/tail manager whenever they change a state variable. The head/tail manager then chooses when to perform the DMA. We decide based on the status of the HOSTEGRESS SRAM queue: if empty, we DMA immediately on the assumption the load is low. If full, the head/tail DMA is done periodically, roughly every 200uS. Avoiding the HEAD/TAIL DMA on every message roughly doubles the DMA bandwidth for small messages. Note, though, that for low latency it is crucial that the UP buffer TAIL be forwarded immediately under low load. The DOWN buffer HEAD is never critical.
3) Finally and most powerfully, we combine small messages for DMA. Each ETHERINGRESS thread decides on a packet-by-packet basis whether to forward its buffer or add another message to it. The buffers are 2KB and the maximum message size is 1522B (1514B plus our 8-byte header). After each packet is received, we add another if (a) another is available and (b) there's room left for a maximum-size packet. Again, the process must be adaptive so that packets are forwarded with low latency under low load.

With the three techniques the net-to-host path provides enough bandwidth to sustain 100% packet throughput on all four ports for all packet sizes (Section III-A).

C. IXP1200 Resources

We estimate the current demand of our design at roughly 40% of the capacity of the chip based on a loose accounting of the processor, memory bus and PCI bus bandwidths. Table I breaks down these estimates by thread. First, we use 50% of the microengine threads. As a capacity, however, this percentage is high because the microengines are not fully utilized. Second, we use 23% and 17% of the boilerplate bandwidth of the SDRAM and SRAM systems, respectively. We regard these percentages as low based on experiments that

	SRAM	SDRAM	PCI
ETHERINGRESS-0	37Mb/S	112Mb/S	–
ETHERINGRESS-1	37Mb/S	112Mb/S	–
ETHERINGRESS-2	37Mb/S	112Mb/S	–
ETHERINGRESS-3	37Mb/S	112Mb/S	–
ETHEREGRESS-0	37Mb/S	112Mb/S	–
ETHEREGRESS-1	37Mb/S	112Mb/S	–
ETHEREGRESS-2	37Mb/S	112Mb/S	–
ETHEREGRESS-3	37Mb/S	112Mb/S	–
HOSTEGRESS	150Mb/S	592Mb/S	448Mb/S
head/tail mngr.	–	–	–
HOSTINGRESS	175Mb/S	200Mb/S	448Mb/S
freelist mngr.	25Mb/S	–	–
	646Mb/S	1688Mb/S	896Mb/S
boilerplate	3712Mb/S	7424Mb/S	2112Mb/S
limits:	17%	23%	42%
practical:	????	4450Mb/S	1600Mb/S
limits:	??%	38%	56%

TABLE I
ESTIMATES OF IXP1200 RESOURCE UTILIZATION VERSUS LIMITS. ALL NUMBERS ARE IN MILLIONS OF BYTES PER SECOND.

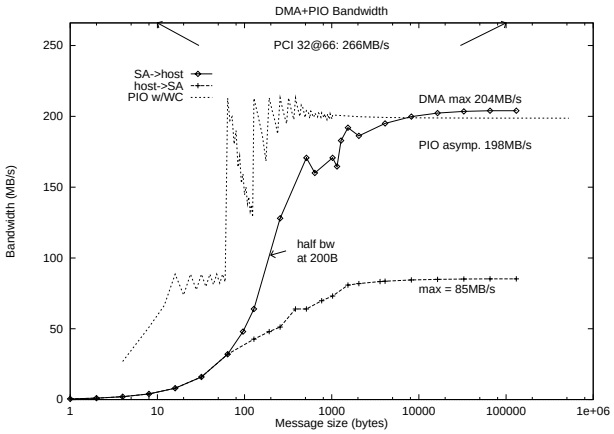

Fig. 3. Bandwidth versus message size for PCI transfers between the IXP ("SA" for StrongARM) and the host. Numbers are in millions of bytes per second. The peaks in the PIO bandwidth correspond to multiples of 64 bytes, the size of the host processor's write-combining buffer.

suggest it is difficult to exploit more than about 60% of the SDRAM bandwidth. Finally, the PCI bus imposes the most serious constraint: we use 42% of the boilerplate bandwidth (42% of 266MB/S) and 56% of the practical bandwidth we have been able to demonstrate (56% of 204MB/S). A full-duplex gigabit interface would push the limits of this chip and it would be difficult to achieve full bandwidth with this PCI card and/or host chipset.

III. RESULTS

In this section, we describe the performance on the network interface in hardware on microbenchmarks. We performed two sets of measurements: the first summarize internal experiments used to tune performance. The second are comparisons to existing 100T and GbE network interfaces using a lightweight messaging system, CLF [6], as a common system.

The experimental platform is a Dell 530 host running linux 2.4.18 with a Radisys ENP2505 card containing the IXP1200. The host has dual 1.7GHz Xeon processors and the Intel 860 chipset. The ENP2505 card has a 64 bit, 66 MHz PCI bus and is plugged into a 64/66 slot in the Dell. The ENP2505 includes an Intel 21555 bridge between the 64/66 external PCI bus and the 32/66 on-card PCI bus of the IXP1200.

A. Internal Experiments

The first set of experiments show internal bandwidth and latency measurement that drove the tuning of the network interface. There are three experiments: PCI bandwidth, end-to-end bandwidth and end-to-end latency. While this chipset and card uses PCI, old technology at this point, it is important to note that the main bandwidth problems arise from the startup latency of the DMA engine, something that would not necessarily change with a newer, higher-bandwidth interconnect. End-to-end, the host-to-host bandwidth ranges from 93% of the "boilerplate" network bandwidth of 400Mbps down to 50% for 64B packets and the latency is 28.3μS host-to-host for a minimum-size packet.

Figure 3 shows PCI bandwidths achieved by the IXP1200's DMA engine in both directions and also programmed I/Os (PIOs) by the host. The buffers are physically contiguous and aligned on 64B boundaries in both the IXP SDRAM and host DRAM. The DMA experiment uses a fixed DMA descriptor and fixed blocks in memory and issues `dma_pci` instructions at maximum rate. The best performance is achieved when the IXP's DMA engine is programmed for a 64B block size and minimum interburst time. The performance for DMA writes to the host is good, 204MB/S out of a maximum 266MB/S, but the performance for DMA reads from the host is poor, only 85MB/S. This poor block read performance is acknowledged in the errata sheet for the 860 chipset [4].

The half bandwidth for DMA to the host is at 200B messages, suggesting that the startup time for DMA is about 1μS per block sent. We experimented with chained DMA descriptors but observed the same performance, suggesting that the bulk of DMA startup time is in fetching and interpreting a descriptor.

PIOs from the host work well with write-combining enabled. For Figure 3, we measured the bandwidth when transferring a long sequence of blocks in a range of sizes. Each block is aligned on 1024B boundaries (the boundary doesn't matter as long as it's a multiple of 64B) and we do an `sfence` operation after each block to force it out of the write-combining buffers. There's a large advantage to transferring blocks that are multiples of 64B – the size of the cache line and of the write-combining buffers in the Xeon. The bandwidth we see with large blocks is 198MB/S.

We get the same bandwidth with word, doubleword (MMX)

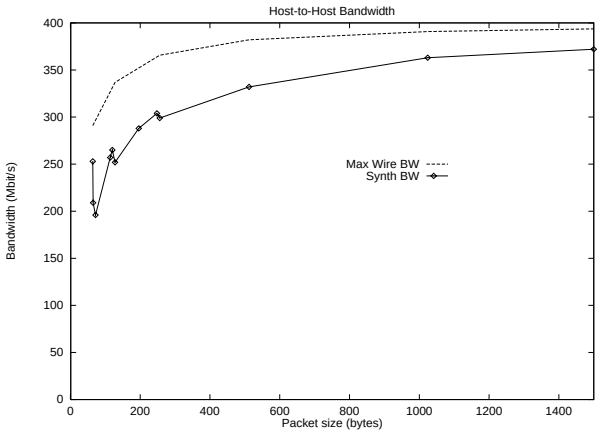

Fig. 4. Bandwidth versus message size for a processor sending packets to itself via all four 100T ports through a switch. Numbers are in millions of bits per second and counts the ethernet packet including 14B header. Peaks occur at multiples of 64 bytes. For reference, the upper line represents the maximum bandwidth versus packet size given 802.3 standard interpacket gap and preamble for 100T ethernet.

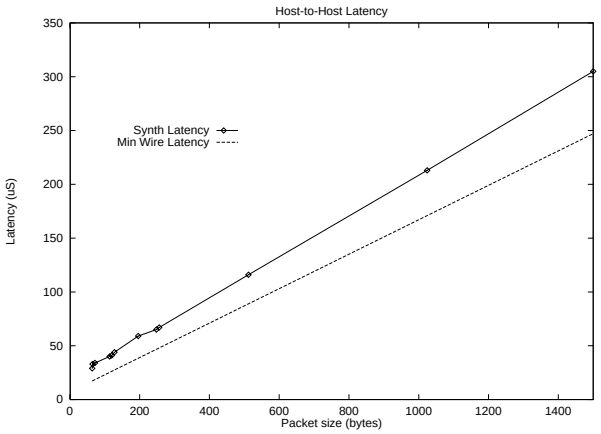

Fig. 5. Latency versus message size for a processor sending packets to itself via all four 100T ports through a switch. The upper line represents user-level host-to-host time. For comparison, the lower line shows the time in the MACs and switch as measured via timestamps in the IXP1200.

or quadword (SSE) writes when write-combining is on. Byte writes, e.g. the standard memcpy, perform poorly. With write-combining off we observe a single-cycle transaction on the PCI bus every six PCI bus cycles and correspondingly poor bandwidth: 11MB/S for byte writes, 44MB/S for word writes and 88MB/S for doubleword/quadword writes.

Figure 4 shows results for host-to-host bandwidth. The experiment uses one host and one IXP board sending messages to itself through all four ports connected to a private 100T switch. Each port sends messages to (port + 1) `mod` 4. The dual-processor host uses two threads: one sending messages and one receiving and checking messages. There is no reliability protocol but no messages are observed to be dropped or corrupted during the test.

The results show bandwidth near the limit imposed by the ethernet protocol, even for small packets. The fact that no messages are dropped indicates that the remaining bottleneck is in the transmit path: the PCI bus (unlikely), HOSTINGRESS or ETHEREGRESS.

Figure 5 shows the host-to-host latency using the same one host, one IXP board setup as the previous experiment. The host ping-pongs a message through the switch 10000 times. We report the time for one round trip – a round-trip in this case corresponds to one host-to-host hop – as measured via gettimeofday() on the host. The lower line in the figure shows the time spent in the MACs and switch as measured via cycle counts on the IXP1200

Table II breaks down latency for the 64B case using data from the cycle counter available on the IXP1200. The majority of the time, 17.7μS, is due to the ethernet hardware: the MACs and the switch (Cisco 3500). The IXP1200 adds 9.3μS in two traversals and the host time is quite small, 1.7μS.

One way to characterize the time spent in the IXP1200 is that it's costing about 2μS per thread along the path (5 threads). While we have not started tuning for latency, it may well be that a lower-latency design will require rethinking the arrangement of threads.

IXP1200 (send)		
HOSTINGRESS	0.9μS	
ETHEREGRESS	2.4μS	
total		3.3μS
100T MACs + switch		17.7μS
IXP1200 (receive)		
ETHERINGRESS	2.6μS	
HOSTEGRESS		
+ head/tail mngr.	3.4μS	
total		6.0μS
Host/PCI (send + receive)		1.7μS
grand total		28.3μS

TABLE II
BREAKDOWN OF ONE-WAY LATENCY FOR A MINIMUM-SIZED (64B) MESSAGE AS MEASURED VIA CYCLE COUNTERS IN THE IXP1200.

B. CLF Comparisons

In the second set of experiments, we compare the IXP-based network interface to 100T and gigabit interfaces using a lightweight message system, CLF, as a common user interface. We do four experiments as tabulated in Table III: CLF over 100T ethernet, CLF over gigabit ethernet, CLF over the IXP1200 set up to look like a standard ethernet device to the kernel and finally CLF using the IXP1200 NI directly from user mode. When the NI is used with the kernel, we enable interrupts in the firmware. The firmware generates an interrupt every time the head/tail manager thread updates the shared buffer state.

Figure 6 shows the results for latency in the four configurations. The experiment is to ping-pong a message 10000 times between two machines and report half a round-trip-time as the latency. The IXP1200 under CLF achieves 38.6μS latency,

	messages	driver	card	medium
CLF	UDP	eepro100	Intel Pro 100	100T
CLF	UDP	(ours)	IXP1200	100T
CLF	raw 802.3	–	Intel Pro 100	4x 100T
CLF	UDP	e1000	Intel Pro 1000	1000F

TABLE III
CLF EXPERIMENTS.

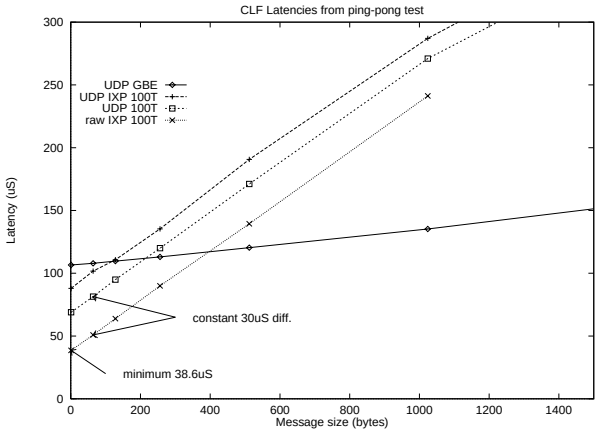

Fig. 6. Latency versus message size between two machines for the four CLF configurations. Numbers are in microseconds.

about 10μS more than the standalone test in the previous section. However, this mode is about 30μS better across the board than the nearest competitor, UDP over 100T ethernet. Using the IXP1200 as a standard ethernet device is worse (no doubt due to our driver) and gigabit ethernet is worst of all. The gigabit path uses a different switch (a Cisco 6500) that we have not yet measured.

Figure 7 shows the results for bandwidth in the four configurations. At the time of writing, they are all similar and quite low. The measured bandwidths correspond closely to an extra 5μS/message for every system, something that likely

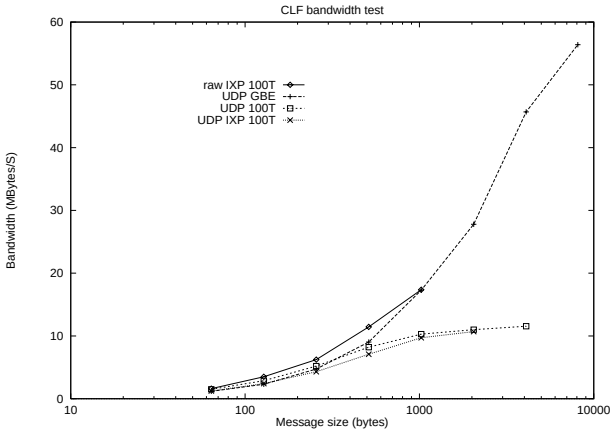

Fig. 7. Bandwidth versus message size between two machines for the four CLF configurations. Numbers are in millions of bytes per second.

represents a bug in our version of CLF. However, the results for the raw IXP1200 interface is encouraging: it tracks the gigabit performance, slightly exceeding it for small packets.

IV. APPLICATION EXTENSIONS

An eventual goal of our project is to incorporate application extensions into the network interface via microengine code and external coprocessors. This section characterizes several network stream oriented applications then characterizes the computation kernels that would execute on the network processor to support those applications. We conclude by discussing where we would implement these extensions in the IXP1200 firmware.

A. Application Scenarios

The target applications share the attributes that they communicate significant amounts of data in small messages and that they require relatively small numbers of operations per byte of data. Consequently, the advantage of performing these computations in the network interface is magnified, since the cost of staging the data through the memory hierarchy would be amortized over relatively few main CPU operations.

Processing on streams at the network interface can reduce work for the host in three ways: by *aggregating* small blocks into larger blocks to reduce per-block overhead, by *classifying* blocks (e.g. to filter out irrelevant messages or to demultiplex messages into classes) and by *reorganizing* data. Reorganizing is open-ended, ranging from simple things like byte-swapping a few fields to the full computation required for that message. We stick with the term "reorganizing" to emphasize the small amounts of computation that are extensions of data movement operations but within the packet. In our system, more ambitious computations are envisioned for the reconfigurable logic component that serves as a coprocessor to the network processor but that is outside this present work.

Finally, there is a fourth opportunity in the network interface and that is to *schedule* messages with sensitivity to the current environment.

The following five applications we have worked with serve as examples of systems that demand small amounts of computation on large numbers of small packets.

1) An information system receives many small update messages from widely distributed sources. The NI can first aggregate these small messages into larger chunks of work to be placed in the server's memory. Second, it can reorganize the messages by stripping headers, correcting byte-ordering or floating-point formats in the payload, uncompressing bitfields, etc.
2) An airline global information system receives a continuous stream of small messages as above from all airports, ticket agents, baggage handlers and the FAA. Many of these messages can be dropped while some are crucially important and deserve to be replicated for high availability. The NI can classify packets using load-sensitive rules to drop messages and to replicate selected messages.

3) A scientific simulation application runs on a server while scientists elsewhere want to visualize results of the simulation in real time as they are generated. The visualization requires a subset of the simulation data. The NI can classify data packets passing through the network and replicate ones of interest to the simulation, then reorganize the data, e.g. by averaging or decimation, before passing it along to the visualization engine.
4) A distributed simulation system such as a distributed interactive game or training environment has the data distribution problem: the problem that each site must multicast information about local events to distant participants. The problem is that the multicast set is both fine-grain and dynamic. In practice the data must be delivered to a superset of recipients which then discard unneeded elements. The network processor can implement classification at both senders and receivers where the filtering rules are updated dynamically.
5) Network monitoring, network statistics collection and intrusion detection all must classify incoming network packets. For instance, consider a network interface that passively detects SYN-flood distributed denial of service attacks on the host computer by continuously estimating the difference between the number of SYN packets received and the number of SYNACKs sent.

From the applications above we see a few kernel operations are implemented repeatedly in the network interface: aggregation, classification, reorganization and scheduling.

B. Results for Kernels

We characterized instances of each type of kernel as they would be implemented in the network interface. These are all data-intensive operations on streams that flow through the network interface in a cluster computer. The instances are not exhaustive but are intended to establish that there are a number of useful operations on streams that require only a few operations per byte of data. While these operations are reminiscent of networking operations (which is part of the reason they are well suited to execution on a network processor), they differ from network operations in that they are invoked from the application level.

Except as noted, the kernels were implemented on the IXP1200 microengines in hand-coded assembly and evaluated using the simulator in Intel's IXP1200 Software Development Kit v1.2 [2] by measuring cycles per iteration for a large number of iterations. The simulator is claimed to be cycle-accurate to within 2%.

1) **Message Filtering.** A message filter evaluates a predicate on a message to determine a course of action for this message. For instance, a visualization application may want to keep only the packets for which the predicate is true or a fault-tolerance application may want to make duplicates of packets for which the predicate is true. We use the visualization scenario as a driver for a representative filter: assume a packet contains an array of 32-bit floating point numbers and define a predicate that is true if a fraction f of the numbers are above a threshold t. The core of this predicate is three instructions (a compare, a branch and an increment) per 32-bit number leading to a computational requirement of 0.75 ops/byte. This kernel was handcoded in IXP1200 assembly and executes at full memory bandwidth on the IXP1200.
2) **Image Downsampling by Averaging.** A video stream can be displayed on a low-performance device if the image is reduced in size and/or framerate. Reducing the framerate is a filtering problem but doing a good job of reducing size requires averaging adjacent pixels. We implement a 4-to-1 reduction for an image encoded as in a 24-bit RGB format. This kernel was not implemented on the IXP but was coded in C using 32-bit operations on a workstation (Sun UltraSPARC using Sun's SPRO 6.1 compiler) so the number of operations should be similar or better for the 32-bit IXP1200 which has more registers and fused ALU-shift instructions. The algorithm converts 24 to six pixels at a time requiring (from inspection of the assembly) 69 operations per 48 input bytes or about 1.5 ops/byte.
3) **Image Depth Reduction.** An alternative method to reduce image size is to reduce the depth per pixel. A 24-bit RGB to 12-bit (4/4/4) RGB reduction requires unpacking and repacking pixels. We handcoded this kernel in IXP1200 assembly and observed that it required about two operations/byte (shift-and-mask, or).
4) **Real-Time Scheduling.** A scheduler must update some form of priority queue for each packet processed and then select the best candidate packet from the queue. The worst case for scheduling is when all packets are minimum-sized. We implemented a Dynamic Window Constrained Scheduler (DWCS) algorithm in assembly on the IXP1200 and observed a cost of about 100 operations per packet. If we assume a minimum packet size of 64 bytes, that's about 1.6 ops/byte.

The kernels above perform useful tasks but require only small amounts of processing ranging from 0.75 to two ops/byte.

C. Implementation of Extensions

The firmware described in Section II permits application code to be attached in any of several possible places: as separate threads, colocated with ETHERINGRESS or colocated with ETHEREGRESS (X-, RxX- or XTx-handlers in the parlance of [3]). Colocating with ingress/egress is desirable for performance but is challenging to program:

- Colocating with ingress/egress can avoid a memory copy. For instance, in a user fragmentation scheme, ETHERINGRESS could determine exactly where in memory to place an incoming packet.
- ETHERINGRESS/ETHEREGRESS operate under hard real time constraints. User code implemented in these threads would be under the same constraints, a significant programming problem.

- Colocating in any particular spot limits functionality. For instance, user code to perform customized multicast (sending slightly different versions of a message) cannot be colocated with reception.

The experiments in [3] suggest that for the IXP1200 it is best to colocate user code with ETHEREGRESS when possible.

V. Related Work

Many network interfaces for clusters and multiprocessors have been built but this is the first we are aware of using emerging network processors. Network processors offer the new possibility of operating on data as it passes through the network interface, not just headers. Network processor work has focused on forwarding in the network, not on network interfaces. Using a network processor as a network interface makes sense for clusters where the bulk of communication problems are at the endpoints.

Processor-based LAN interface cards such as the Myrinet LANai card [5], Alteon AceNIC card [1], Intel I2O card, etc. and network interface coprocessors in multiprocessors, e.g. the Intel Delta/Paragon [8], Meiko CS-2 [9], etc. have been user-reprogrammable but the processors are notably slow compared to the data rates. Network processors are provisioned to operate at line rate on minimum-sized packets and thus offer the opportunity to operate on more than just packet headers. We measure the headroom on the IXP1200 at about three 32-bit ops/byte when moving data unidirectionally at gigabit speed. The extensive multithreading on the IXP1200 and other network processors offers new design opportunities and challenges as well. Network processor work has focussed on traditional uses in networking, e.g. forwarding [10], [11].

VI. Conclusion

We have presented a network interface based on an Intel IXP1200 network processor, described the design and presented performance measurements based on microbenchmarks. The IXP1200 is on a commercially available PCI card. Our microbenchmark performance is near hardware limits and CLF performance is encouraging.

Acknowledgment

The authors would like to thank the rest of the ASAN group for many useful ideas and feedback. This work was funded in part by the National Science Foundation under grant # ANI-9876573, in part by the U.S. Department of Energy under contract # DE-FC02-99ER25402 and in part by the State of Georgia's Yamacraw Mission. Equipment used in this work was donated by Intel Corporation.

References

[1] Alteon Web Systems. `http://www.alteonwebsystems.com/-products/acenic/`.

[2] Intel Corportation. IXP1200 Software Development Kit, v2.0, March 2001. `http://developer.intel.com/design/network/-products/npfamily/`.

[3] Ada Gavrilovska, Kenneth Mackenzie, Karsten Schwan, and Austen McDonald. Stream Handlers: Application-specific Message Services on Attached Network Processors. In *Hot Interconnets: 10th International Symposium on High Performance Interconnects*, August 2002.

[4] Intel. *Intel 860 Chipset: 82860 Memory Controller Hub (MCH) – Specification Update*, May 2001.

[5] Myricom Inc. `http://www.myri.com/`.

[6] Rishiyur Nikhil. *Cluster Language Framework (CLF) Version 2.10*. DEC CRL, November 1997.

[7] Radisys. *ENP-2505 Hardware Reference*, November 2001.

[8] Rolf Riesen, Arthur B. Maccabe, and Stephen R. Wheat. Split-C and Active Messages under SUNMOS on the Intel Paragon. Unpublished, April 1994.

[9] Klaus E. Schauser and Chris J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Symposium on Parallel Processing*, 1995.

[10] Tammo Spalink, Scott Karlin, and Larry Peterson. Evaluating Network Processors in IP Forwarding. Technical Report TR-626-00, Department of Computer Science, Princeton University, November 2000.

[11] Tammo Spalink, Scott Karlin, Larry L. Peterson, and Yitzchak Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Eighteenth ACM Symposium on Operating Systems Principles*, December 2001.