A Reconfigurable Extension to the Network Interface of Beowulf Clusters*

Keith D. Underwood† Ron R. Sass Walter B. Ligon, III

Parallel Architecture Research Lab
Holcombe Department of Electrical and Computer Engineering
Clemson University
105 Riggs Hall
Clemson, SC 29634-0915
{keithu, rsass, walt}@parl.clemson.edu

Abstract

With a focus on commodity PC systems, Beowulf clusters traditionally lack the cutting edge network architectures, memory subsystems, and processor technologies found in their more expensive supercomputer counterparts. What Beowulf clusters lack in technology, they more than make up for with their significant cost advantage over traditional supercomputers. We propose an architectural extension that adds reconfigurable computing to the network interface of Beowulf clusters. This enhances both the network and processor capabilities of the cluster. Furthermore, for some applications, the proposed extension partially compensates for weaknesses in the PC memory subsystem. We discuss two applications, the 2D Fast Fourier Transform (FFT) and integer sorting, which benefit from the resulting architecture.

1. Introduction

Beowulf-class computers (cluster computers based on COTS hardware and open system software) have emerged as a cost-effective supercomputing solution for a variety of problems. Unfortunately, the COTS hardware that enables cost-effectiveness also limits the applicability of Beowulf clusters to classes of problems, such as those similar to the Fast Fourier Transform. In general, problems that depend on a high performance interconnect to achieve scalability tend to scale poorly. Beowulf clusters generally lack the cutting edge network, memory, processor, and I/O subsystems found in their more expensive supercomputing counterparts.

*This work was supported in part by the National Science Foundation under NSF Grant EIA-9985986

†Mr. Underwood is supported by a NASA GSRP Fellowship under grant number NGT5-85

We propose a new architectural feature that integrates reconfigurable computing (RC) with a standard high performance network interface to form an Intelligent Network Interface Card (INIC). By placing RC in the network datapath, the INIC offers a mechanism to offload application processing and network processing from a node's processor(s). Furthermore, it helps hide the weak PC memory hierarchy by providing extensive capabilities to manipulate the data stream. This is expected to deliver significant gains in performance.

Though reconfigurable technology is currently impractical due to cost, recent advances have created an opportunity for a commodity implementation. That analysis, however, is beyond the scope of this paper (please see [17] for discussion of cost). Instead, this paper focuses on a description of the proposed architecture (Section 2) and an analysis of its performance (Section 4). We use two applications (FFT and integer sorting) described in Section 3 to highlight the performance of our architectural extension. In Section 4, we model the performance of our proposed system for these two applications, highlighting the source of our performance gains. To validate our analysis, we have built an Adaptable Computing Cluster (ACC) — a prototype 8-node cluster utilizing an off-the-shelf RC card fitted with a Gigabit Ethernet NIC. This prototype cluster is described in Section 5. We compare our INIC prototype against ordinary Gigabit Ethernet in Section 6. The paper is concluded with a discussion of related work (Section 7) and a summary (Section 8).

2. ACC System Architecture

The goal of our ACC project is to explore architectural enhancements to Beowulf clusters. In particular, we aim to broaden the class of applications that map well to this type of cluster. Figure 1 illustrates the core of our proposed cluster enhancement. Whereas a traditional NIC simply buffers

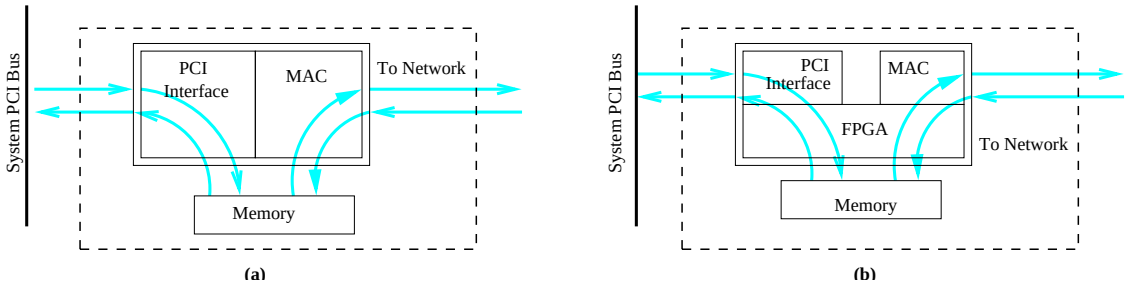

Figure 1. (a) comparison of a traditional NIC and (b) the proposed architectural extension to form an INIC

data in a memory between the host and network, we propose inserting FPGA-based RC along the datapath to form an INIC. The FPGAs could then be used in a range of modes including:

- *Compute Accelerator* — Defined as using the FPGAs strictly for application computing tasks, this mode significantly enhances the computing power of a node. A cluster with reconfigurable computing at every node, such as the Tower of Power [13], amplifies this capability. When using the INIC for compute acceleration, a separate path to host memory is configured to allow normal network operations.

- *Protocol Processor* — As a protocol processor, the FPGAs are used strictly for network processing. A properly designed Intelligent NIC could perform all of the protocol processing for a node, offering more features (such as collective operations), higher bandwidth, and lower latency than current commodity network subsystems. An INIC can always be designed for better network performance than current systems because the intelligence can be coupled with any network fabric. (Unlike some solutions that have attempted to use an embedded processor on the NIC for protocol processing, reconfigurable computing provides more than enough computing power for full rate transfers at any data rate.) The performance boost comes from having protocol processing on the card and eliminating the need for frequent, per-packet interrupts. Further, acknowledgement packets and per packet protocol overhead need not consume system bandwidth (bus or memory).

- *Combined Compute/Protocol Accelerator* — Placing computing and protocol elements in the FPGAs takes advantage of the opportunity to tightly couple a high-performance computing core with the network interface. This is the most interesting of the three modes as it has the advantage of very low latency from the computing core to the protocol accelerator. Alternatively, the computing portion can be a passive element, processing data as it passes through the device at zero cost.

This emphasizes the architecture's ability to improve the communication and computation performance of each node in a cluster. More significantly, the project's goal is to show that the introduction of an INIC does more than just add RC or enhance networking. Rather, the two enable each other to succeed in improving applications that neither technology alone improves.

3. Applications

Application performance is the metric for evaluating new architectural ideas. Two applications have been chosen for the preliminary evaluation of the INIC architecture: the Fast Fourier Transform (FFT) and integer sorting. These applications were chosen as examples of an INIC's ability to absorb a significant amount of an applications into the communications operation. For each application, the architecture-specific implementation is derived from the standard parallel implementation. This section discusses the applications and their implementation. Diagrams illustrating the implementations use a rounded box to represent processes, rectangles for hardware function blocks, and arrows to indicate data flow. Where there is ambiguity, sequences are numbered. Each diagram shows one of many concurrent network transactions in the cluster.

3.1. 2D-FFT

The 2D-FFT is used in a number of applications and is representative of a broader class of parallel workloads that must manipulate matrices that are contained in distributed memories. The fundamental equation is

$$Y[i_1, i_2] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} X[j_1, j_2] \omega_1^{-i_1 j_1} \omega_2^{-i_2 j_2}. \quad (1)$$

This is commonly grouped as

$$Y[i_1, i_2] = \sum_{j_2=0}^{n_2-1} \left(\sum_{j_1=0}^{n_1-1} X[j_1, j_2] \omega_1^{-i_1 j_1} \right) \omega_2^{-i_2 j_2} \quad (2)$$

revealing the standard parallel implementation described below.

3.1.1. Parallel Implementation

Our baseline parallel implementation is the highly optimized Fastest Fourier Transform in the West (FFTW) package[12]. On a distributed memory machine, such as a cluster, FFTW distributes a block of the rows of the data to each processor. Then, the algorithm is parallelized as the following four steps.

❶ compute the 1D-FFT for each row

❷ transpose the matrix (redistribution of data)

❸ compute the 1D-FFT for each row

❹ transpose the matrix (a second redistribution of data)

3.1.2. ACC Implementation

Implementation of the FFT on an ACC follows the same template as FFTW. The matrix transpose at the core of the algorithm couples extensive data reorganization with network operations. Since the matrix transpose is a serialized communications step, it becomes the limiting factor for parallel scaling of the FFT. Because of this, the reconfigurable resources on the INIC are best dedicated to optimizing the transpose. Like FFTW, an ACC implementation decomposes the transpose into three components: a local transpose step, an all-to-all communication, and a final permutation. With a row block data distribution, each processor has M rows of data, where $M = rows/P$ and P is the number of processors. Assuming a square matrix is being transformed, each processor exchanges an $M \times M$ block with every other processor. To perform a transpose we locally transpose each block, send the block to the appropriate node, and interleave data from the blocks as they are received.

Unlike the standard parallel implementation, an implementation for the ACC pushes all data manipulation needed for the transpose (on both send and receive side) onto the INIC, as shown in Figure 2. This allows the data manipulation to be embedded in the communication at minimal additional cost. Furthermore, the communication protocol used can be customized to the specific application since each node knows exactly how much data will be sent to and received from every other node.

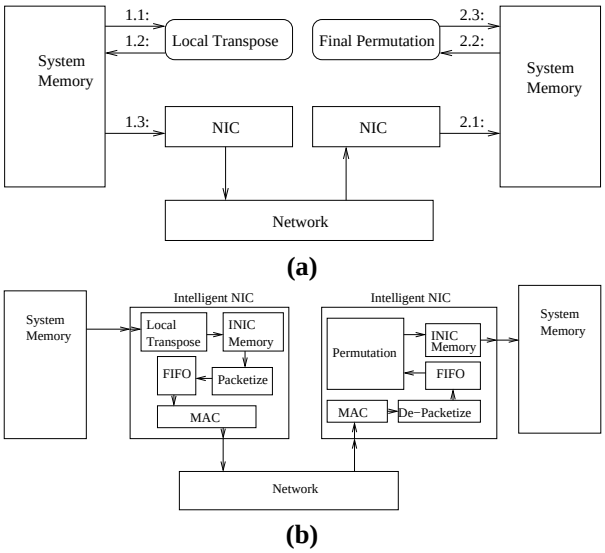

Figure 2. (a) block diagram of the standard parallel transpose algorithm; (b) INIC manipulation of data stream

3.2. Integer Sorting

Integer sorting is a classic benchmark application. Although some sorting benchmarks involve non-uniformly (often Gaussian [2]) distributed data, our input data is synthetically generated and uniformly distributed. Although this is not a realistic assumption, it is a well-established precedent. Using uniformly distributed keys allows us to focus on evaluating the basic I/O and computational performance of our architecture and, perhaps more importantly, permits our results to be be compared directly with previously reported numbers (presented by researchers using the same assumptions). As others have recognized, sampling in a pre-sort phase helps address the shortcomings of our assumption by leading to a more balanced workload.

Each of our sorting implementations are based on the Count Sort as in [1]. We found that Count Sort was as much as $2.5\times$ faster than quicksort. We also found that in our test environment it is important to first bucket sort the data such that the buckets fit in the processor cache.

3.2.1. Parallel Implementation

The parallel implementation of the distributed memory to distributed memory sort assumes the data is initially distributed across P processors where P is a power of two. Each processor begins by bucket sorting its data into P buckets. Bucket i from each processor is then sent to processor i.

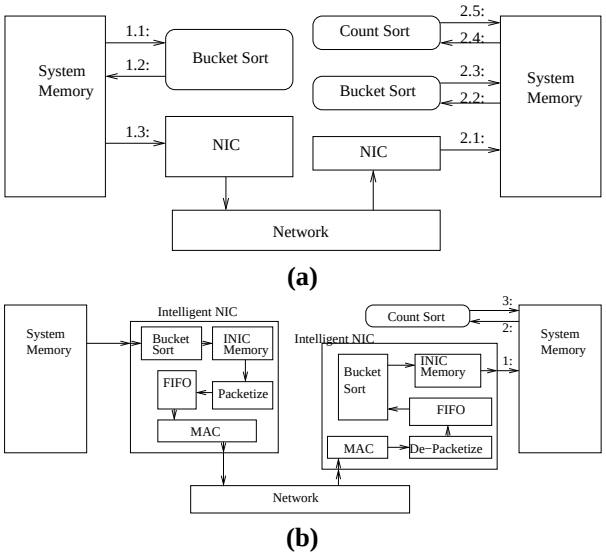

Figure 3. (a) operations involved in a standard parallel integer sort; (b) bucket sorting in INIC implementation

As a processor receives the data, it bucket sorts the data into buckets designed to fit in processor cache. Once all data has been collected, each bucket is sorted with Count Sort. The Count Sort is the final sorting phase — with 32 bit integers and more than 128 buckets there is no need for the final bubble sort described in [1]. On a problem size of 2^{21} keys or more, a minimum of 128 buckets are needed for the problem to map well into cache.

3.2.2. ACC Implementation

Figure 3 illustrates the differences in data flow for a traditional parallel implementation and an ACC implementation. Like the parallel implementation, the ACC implementation must bucket sort the data on the sending and receiving nodes. Since bucket sorting is particularly amenable to implementation on the INIC, we implement both bucket sorts in hardware. Three factors make the bucket sorts a good choice for implementation on the INIC. The bucket sorts: (i) are tightly coupled to communications, (ii) reorganize the entire data stream in a single pass, and (iii) compose a significant portion of the computational load.

Figure 3(b) diagrams these operations implemented in the INIC. On the sending side, data is transferred directly from host memory to INIC memory. Along the path, the data is manipulated to perform the bucket sort operation. Like the parallel implementation, the INIC implementation can overlap communication with computation. In fact, the INIC can start transmitting data at lower bucket thresholds (one packet) since there is no computational cost for starting a send. (This is not to say that there is no computation involved in starting a send, only that starting a send is handled by hardware that sits idle if no send is in progress.) Hence, on the sending side, the INIC handles all of the computation and protocol processing leaving the processor free to other tasks such as disk I/O.

On the receiving side, the bucket sort can be done as data is received. As minimum thresholds are reached, data is transferred to the host. After all data is received, each bucket is sorted with Count Sort. While at first glance it might appear that Count Sort would be amenable to implementation in hardware, it should be noted that cache memory bandwidth on a commodity processor is much higher than the comparable memory bandwidth for an INIC, making certain cache-friendly, memory-intensive operations better suited to the host processor.

4. Architecture Analysis

Here, the potential performance of the proposed INIC with a Gigabit Ethernet interface is analyzed and compared to the performance of a cluster using a Gigabit Ethernet NIC. Although an INIC is not limited to gigabit per second networks, the same underlying network is kept for comparison. While the results presented in this section are theoretical, they are supported by preliminary measurements from our prototype. Individual bandwidths used in the analysis have been measured, including the bandwidth to and from the INIC while performing the local transpose steps. INIC to INIC bandwidth has also been measured. Numbers used in calculations are a conservative 80%–90% of measured results.

4.1. FFT Performance

The run-time, T, of the FFTW application can be approximated as the sum of the time to compute the FFT and the time to transpose the matrices, or

$$T = T_{compute} + T_{trans} \tag{3}$$

with $T_{compute}$ defined as the time to perform the FFT computation for each row assigned to the processor. This can be written as,

$$T_{compute} = 2 \times \left(T_{\text{1D-FFT}}\left(rows\right) \times \frac{rows}{P} \right) \tag{4}$$

where $T_{\text{1D-FFT}}\left(rows\right)$ is the time to compute the 1D-FFT of one row of a matrix of size $rows$, $rows$ is the number of rows in the (presumably square) matrix, and P is the number of processors used. The remainder of the application

time is accounted for in T_{trans}, which is the total time spent doing the two required matrix transposes. For an INIC, the transpose time can be defined as follows. The partition size, S, (in bytes) is,

$$S = \frac{rows^2 \times 16}{P} \tag{5}$$

where 16 is the number of bytes to store a complex double precision element and P is the number of processors. Assuming that we can pipeline data movement from host to card and from card to network, transferring the data to the FPGA memory from host memory requires

$$T_{dtc} = \frac{\frac{S}{P}}{80 \times 1024 \times 1024} \tag{6}$$

seconds of delay. The transfer of data from FPGA memory to the network requires an additional

$$T_{dtg} = \frac{\frac{S}{P}}{90 \times 1024 \times 1024} \tag{7}$$

seconds of delay. Again, we assume that receives can be pipelined with sends after each card has transmitted one processor's worth of data. Insuring that each processor is always sending and receiving is a fairly simple matter. The time (in seconds) to receive the data from the network can be described by:

$$T_{dfg} = \frac{\frac{(P-1)\times S}{P}}{90 \times 1024 \times 1024}. \tag{8}$$

The final copy of data to the host must wait on all data to be received, so the time required for the host to retrieve the results is:

$$T_{dth} = \frac{S}{80 \times 1024 \times 1024} \tag{9}$$

Thus, the total time to perform both matrix transposes is the sum of (6) through (9) times 2 because the transpose is done twice (steps ❷ and ❹ in Subsection 3.1),

$$T_{trans} = 2 \times (T_{dtc} + T_{dtg} + T_{dfg} + T_{dth}) \tag{10}$$

Based on this analysis, we have created the graphs shown in Figure 4.

Looking at Figure 4(a), we see that even though they use the same network technology, an Intelligent Gigabit Ethernet NIC significantly outperforms the standard Gigabit Ethernet NIC for a matrix transpose. To understand why, we need to look inside the numbers. Since the only difference in the implementations is the transpose, the graph in Figure 4(b) shows a decomposition of the transpose time for the Gigabit Ethernet NIC and INIC. The compute time scales as expected. The curve is smooth except at 2–3 processors and 6–8 processors where the local partition fits into a faster level of the memory hierarchy. Communication time, however, does not scale as well. Since the Intelligent NIC numbers are estimates, one might argue that real measurements would be significantly worse; however, it is important to note that the poor scalability seen in the communication time for the Gigabit Ethernet NIC is a characteristic of the TCP/IP protocol and the PC system architecture.

First, TCP is designed for long, moderate-latency, low bandwidth transmissions over a lossy, and possibly congested, channel. A cluster typically uses a low latency, high bandwidth, extremely low loss, limited congestion channel. Furthermore, as we move to the right in the graph in Figure 4(b), the data size exchanged between processors becomes relatively small. To complicate matters further, high speed network interfaces typically use some form of interrupt mitigation — based on a time-out or number of messages received. This mechanism is necessary because modern systems are incapable of handling an interrupt per packet at the full data rate of Gigabit Ethernet, but it interacts poorly with TCP slow-start for short messages. These factors combine to contribute to network overhead and to prevent the transpose communication time over Gigabit Ethernet from decreasing as rapidly as the partition size. This is evident in the graphs as the line representing partition size has a steeper slope than the one representing communication time.

The intelligent NIC does not suffer from these problems. The foremost improvement is the virtual[1] elimination of interrupts from the communication path. The FPGAs are able to respond instead to the memory accesses by the Gigabit NIC on the PMC slot. Beyond that, INICs can use an application specific protocol. In this particular instance, there should be no packet loss as the total amount of data put into the network never exceeds the total size of the network buffers (combined NIC and switch buffers). The protocol also has the advantage of knowing exactly how much data to expect; hence, the protocol needs minimal acknowledgement information.

Referring back to Figure 4(a) we see near linear speedup for our INIC based system. The graphs show good scalability with no substantial indication of when that linear speedup will end. The FFTW application as implemented with an INIC will stop scaling when communication time no longer scales down with partition size. This could be caused by any number of system issues. On the PC architecture, it is likely to be the limits on the efficiency of the DMA engines at transferring data to and from host memory.

[1] Initiation of the transfer of data to the host memory may require a single interrupt per transpose. On rare occasion, interrupts may be needed for error handling

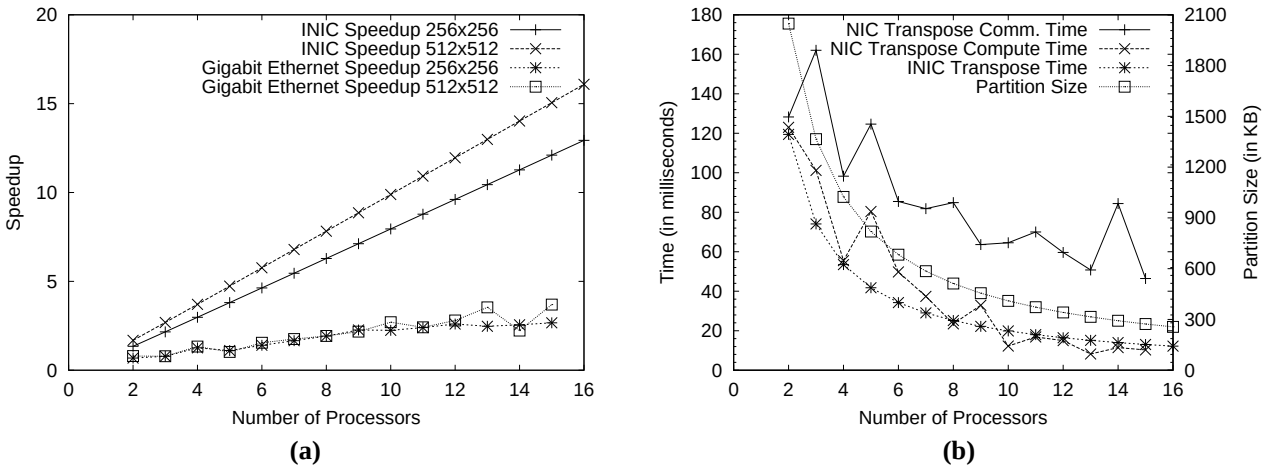

Figure 4. (a) FFTW speedups for an Intelligent NIC and a cluster based on Gigabit Ethernet; (b) a decomposition of time spent in each phase of the operation compared to partition size

4.2. Integer Sort Performance

Since the integer sort application sorts data from distributed parallel memory to distributed parallel memory, the run-time, T, can be approximated as the sum of the time to redistribute the data and the time to perform the count sort on the final buckets, or

$$T = T_{countsort} + T_{INIC} \tag{11}$$

with $T_{countsort}$ being dependent on the number of elements on each processor and thus the same for any of our implementations. A graph of measured $T_{countsort}$ for various numbers of processors is shown in Figure 5(a).

The remainder of the application time is accounted for in T_{INIC} which is the total time from when the data enters the sending INIC until it is retrieved from the receiving INIC. T_{INIC} is dependent on the partition size (elements per processor), the size of packets used to communicate, and the transfer size on the receiving host.

The partition size, S, (in bytes) is,

$$S = \frac{4 \times E_{init}}{P} \tag{12}$$

where 4 is the number of bytes to store an integer, E_{init} is the total number of elements being sorted, and P is the number of processors. Assuming that we can pipeline data movement from host to card and from card to network, transferring the data to the FPGA memory from host memory requires

$$T_{dtc} = \frac{P \times 1024}{80 \times 1024 \times 1024} \tag{13}$$

seconds of delay. This assumes a packet size of 1024 bytes and assumes the worst case distribution of data into bins before transmits can begin. A packet size of 1024 is reasonable since each design can have a protocol built directly on Ethernet. This minimizes overhead in the packets. Further, since our architecture eliminates interrupts and does not involve a shared bus between the NIC and the reconfigurable logic, there is no particular incentive to maximize the packet size.

The transfer of data from FPGA memory to the network requires an additional

$$T_{dtg} = \frac{P \times 1024}{90 \times 1024 \times 1024} \tag{14}$$

seconds of delay. Again, we assume that receives can be pipelined with sends, hiding additional delay. The delay (in seconds) for receiving the data from the network can be described by:

$$T_{dfg} = \frac{N \times 65536}{90 \times 1024 \times 1024} \tag{15}$$

where 64 KB is the minimum size transferred from the card to host memory to ensure efficiency of the DMA operation. We must receive N times this size, where N is the number of buckets we are sorting into on the receiving side, before being guaranteed that any one bucket will cross the threshold for transfer. Finally, the time required for the host to retrieve the results is:

$$T_{dth} = \frac{S}{80 \times 1024 \times 1024} \tag{16}$$

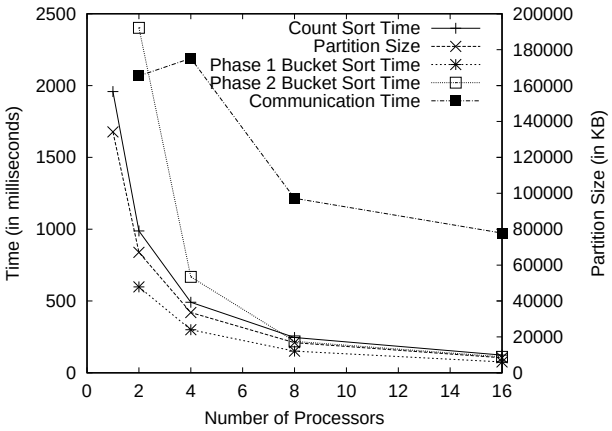

Figure 5. (a) Count Sort time, phase 1 bucket sort time, phase 2 bucket sort time, communication time, and partition size vs. number of processors with sort times on the left axis (ms) and partition size on the right axis (KB); (b) comparison of integer sort parallel speedups for INIC and Gigabit Ethernet implementations

Thus, the total time to perform the data redistribution is the sum of Equations (13) – (16),

$$T_{INIC} = T_{dtc} + T_{dtg} + T_{dfg} + T_{dth} \quad (17)$$

Figure 5(b) compares the speedups of an INIC and a Gigabit Ethernet implementation of integer sorting. The superlinear speedups achieved by the INIC implementation is attributable to the elimination of the time for bucket sorting the data (over 5 seconds in the serial implementation). The INIC exhibits much better speedups than the Gigabit Ethernet solution by eliminating bucket sorting time and implementing a better protocol for the relatively small data transfers caused by larger numbers of processors. It is difficult to directly compare the communications times of the two implementations because bucket sorting time is overlapped with communication times for best performance in a Gigabit Ethernet implementation. Figure 5(a) breaks out the timed components of a serialized parallel implementation on Gigabit Ethernet.

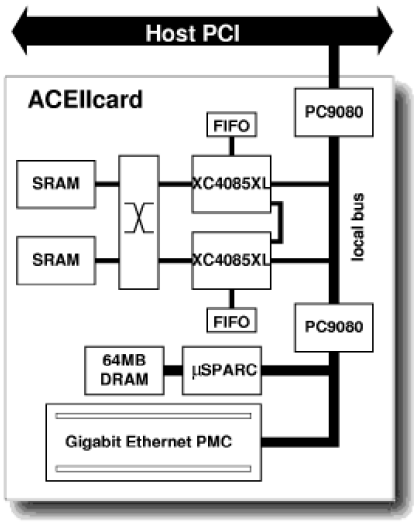

Figure 6. block diagram of an ACEII card

5. Prototype System

Our experimental platform is a 16-node Beowulf running the Scyld Linux distribution. Each node contains a 32-bit PCI motherboard with a 1GHz Athlon and 512 MB of RAM. On the PCI system bus is a SysKonnect PCI Gigabit Ethernet NIC, and a Fast Ethernet NIC. Eight of the systems include an ACEII card (shown in Figure 6) populated with a Gigabit Ethernet NIC.

Although not ideal, the board is a sufficient experimental testbed for prototyping an Intelligent NIC. Furthermore, because it was an off-the-shelf design, we were able to contain costs. It has the added advantage of a PMC connector that offers a standard interface for which a variety of network adapters are commercially available.

Ideally, an Intelligent NIC would be implemented as a single chip with external RAM — similar to modern high performance NICs. Further, the system PCI bus would be sufficient (64-bit 66MHZ or, in the future, PCI-X) to deliver the full bi-directional bandwidth of the network. Architec-

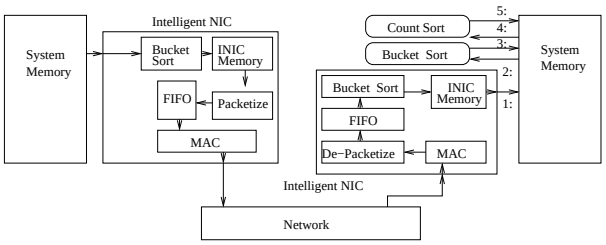

Figure 7. prototype INIC integer sort implementation

tural deficiencies in the prototype prevent it from achieving its full potential as an Intelligent NIC. These deficiencies include a single bus on the card for all data traffic, a 32-bit 33MHz PCI busses, an older generation of reconfigurable logic, and limited memory attached to the FPGAs. Nonetheless it suffices to demonstrate the concepts we are addressing. Newer boards with similar cost address some of these issues but lack the PMC connector necessary to support the networking functions. Section 4 predicted the performance achievable with a next generation Intelligent NIC, while Section 6 addresses the performance achievable with out prototype INIC.

6. Experimental Results

This section compares measured results for our baseline parallel implementations with those achievable on our prototype hardware. Figure 2(b) shows the actual transpose implementation achievable in hardware, and Figure 7 shows the integer sort implementation achievable. For the FFT, the design does not change when compared with with the ideal implementation, but the prototype hardware does introduce a bottleneck in the form of a single 132 MB/s bus used to access both the Gigabit Ethernet and host memory. The integer sort implementation has changed significantly from the one found in Figure 3(b). In addition to the bottleneck introduced by the bus, the Xilinx 4085XLA devices we have are not dense enough to perform the full bucket sort on the INIC. Consequently, the bucket sort must be performed in two phases. The card sorts the data into 16 buckets and the host sorts each of those buckets into N buckets, where N is based on the data size. Surprisingly, this can provide higher performance than having the host sort directly into $16 \times N$ buckets. Despite the limitations of the prototype hardware, it is capable of significant speedups for both applications.

6.1. FFT Results

Figure 8(a) compares the scalability of FFTW on Fast Ethernet, Gigabit Ethernet, and our prototype Intelligent NIC. Intelligent NIC speedups are conservative estimates. These estimates combine measurements of actual times to transfer data to and from the card while performing the local transpose and final permutation based on the proposed design. They also include preliminary bandwidth measurements between two cards. The equation used is similar to the one developed in Section 4 with adjustments to account for the architectural weaknesses of the prototype. Gigabit Ethernet and Fast Ethernet results are measurements taken with the FFTW package. The graph tells an interesting story.

The Fast Ethernet data in Figure 8(a) indicates that the bandwidth of Fast Ethernet severely limits scalability for problem sizes of interest. To exceed the performance of a single processor, a minimum of eight nodes is required, and using 14 nodes barely doubles the performance of a single processor. In contrast, Gigabit Ethernet gives a speedup of two with 8 processors and is able to approach a speedup of four in some cases. This is better, but would hardly be considered scalable. Finally, the prototype INIC performance data indicates that the prototype INIC offers both a significant speedup over and better scalability than the standard Gigabit Ethernet solution, although both use the same network hardware.

Figure 8(b) directly compares the transpose times[2] for the two matrix sizes under consideration using a Gigabit Ethernet implementation and an INIC solution. We can see significant speedup for the matrix transpose operation is given by the INIC based system. Additionally, the slopes of the graphs indicate better scalability from an intelligent NIC.

6.2. Integer Sort Results

Figure 8(b) compares the speedup of our prototype INIC and Gigabit Ethernet implementations. Speedups for the INIC are estimates formed in the same way as those for the FFT. Speedups obtained by the Gigabit Ethernet are measured results. The prototype INIC can not achieve the full potential of the INIC limited both by the bus bandwidth on the card and the need to perform a second stage bucket sort on the receiving host.

[2]Transpose time measurements for numbers of processors that are not a power of two for the INIC are estimates added strictly to smooth the curve. The general shape of the curve made this desirable for clarity.

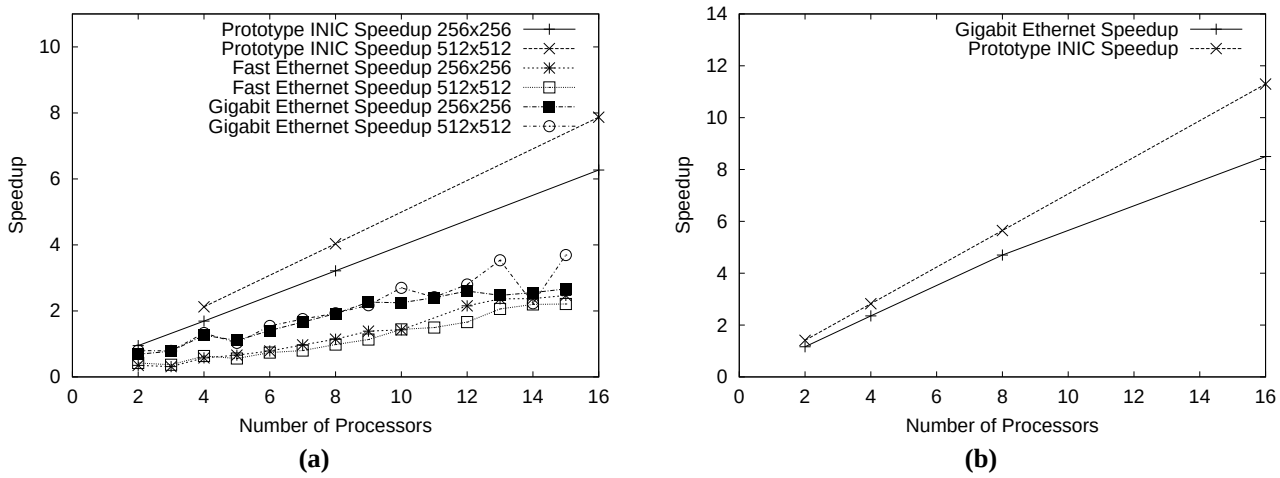

Figure 8. (a) 2D-FFT parallel speedup for a on three technologies: Fast Ethernet, Gigabit Ethernet, and an Intelligent NIC; (b) Integer Sort parallel speedup on Prototype INIC and Gigabit Ethernet

7. Related Work

Until recently, commodity network interfaces and switches had limited bandwidth and high latencies compared to multicomputer interconnection networks. To enhance performance, advances in user-level interfaces and related research has reduced latency. Examples include VIA [5] and its predecessors (AM [19], U-Net [18]). However, software cannot compensate for the bandwidth difference between commodity networks and typical multicomputer interconnects. Thus, efforts like the SHRIMP multicomputer [3] combined commodity-PC workstations with custom NICs and an Intel Paragon routing backplane. Similarly, the Berkeley NOW [9] (and many clusters since) have used Myrinet [4] components or Compaq's Servernet-II [7].

With the advent of competitors to Myrinet, including Gigabit Ethernet and Scali Computing's Wulfkit [8], and considering current prices, it appears that gigabit networks are reaching commodity prices. Thus, the question becomes how best to use this bandwidth. In early distributed memory multicomputers, many designs included a processor dedicated to communications processing. Since then, others have considered using all processors for both communication and computation. In the cluster context, this translates into multiprocessor nodes (SMP workstations). However, results from researchers at the University of Wisconsin [10] suggest that fixing one processor for communication processing benefits light-weight protocols and improves performance when communication is a bottleneck. These results when combined with the arguments made in [11] (which suggest that I/O system busses will continue to inhibit gigabit networking) leads one to focus on adding more processing capabilities to the NIC.

Indeed many, gigabit networks have embedded processors on the NIC that researchers are exploiting in many ways. In this way, we are similar to Typhoon [14], Georgia Tech's VCM [15], RWCP's GigaE PM project [16], and the University of British Columbia's GMS-NP project [6]. All of these use a processor on the NIC to accelerate distributed computing. However, these solutions (1) are based on embedded processors with a fraction of the computing power of reconfigurable logic and (2) ignore the potential of adding application-specific computation on the NIC-side of the I/O bus. Our architectural extension is also inherently different in that the network data stream passes through the reconfigurable logic allowing it to apply an arbitrary dataflow algorithm to the data. Embedded processors are fundamentally incapable of this.

Others have created clusters with reconfigurable cards in each node [13]. However, relatively low PCI bus speeds have always hindered RC and this problem is further complicated when the PCI bus is shared with cluster network traffic. Avoiding this by integrating the RC with the NIC is an important innovation.

8. Summary

We have proposed an architectural extension to the classic Beowulf cluster architecture in the form of reconfigurable computing in the network datapath. We hypothesized that such an extension would provide scalability to applications that have typically mapped poorly to Beowulf clusters.

We discussed two such applications, FFT and integer sort, and their performance on an Adaptable Computing Cluster.

FFT is representative of a broad class of algorithms that are difficult to parallelize on commodity distributed memory clusters because the parallel run-times are dominated by communication operations. As a complex double precision floating point calculation, it is also in a class of applications that has been ill-suited to FPGA technology. Properly merging the two technologies, however, enables significant parallel speedup and drastically better scalability on a commodity cluster. We have illustrated that even with prototype hardware, the INIC is capable of providing significantly better performance.

Integer sorting, on the other hand, is easily parallelized for large problem sizes; however, like the FFT, it performs large data reorganizations in association with the communication operations. Because of this, an INIC offers it a significant performance win. Even with prototype hardware that is unable to perform the full bucket sort on the receive side, the partial bucket sort can improve memory access patterns enough for a performance improvement.

We have shown that these applications receive significant benefits from an Intelligent NIC. Indeed, inserting reconfigurable computing in the path from network to memory has a myriad of benefits, including essentially eliminating all interrupts in many cases. By combining Beowulf clustering and reconfigurable computing, we have achieved a result that is greater than the sum of its parts. The implications of this architecture are far reaching, with the potential to accelerate functions ranging from collective operations to MPI derived data types at a cost that is not prohibitive to commodity implementations.

References

[1] R. C. Argarwal. A super scalar sort algorithm for RISC processors. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 240–246, June 1996.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA, Dec. 1995.

[3] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, and E. Felten. Virtual memory mapped network interface for the SHRIMP multicomputer. In *21st Annual International Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1), Feb. 1995.

[5] P. Buonadonna, A. Geweke, and D. E. Culler. Implementation and analysis of the virtual interface architecture. In *Proceedings of the 1998 SC Conference*, Nov. 1998.

[6] Y. Coady, J. S. Ong, and M. J. Feeley. Using embedded network processors to implement global memory management in a workstation cluster. In *Proceedings of The Eighth IEEE International Symposium on High Performance Distributed Computing*, Aug. 1999.

[7] Compaq. Compaq Servernet II SAN interconnect for scalable computing clusters, June 2000. From Whitepaper found at http://www.compaq.com/support/techpubs/whitepapers/-tc000602wp.html.

[8] S. Computing. Wulfkit, Jan. 2001. From webpage at http://www.scali.com/.

[9] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel computing on the Berkeley NOW. In *9th Joint Symposium on Parallel Processing*, 1997.

[10] B. Falsafi and D. A. Wood. Scheduling communication on an SMP node parallel machine. In *Proceedings of Third International Symposium on High Performance Computer Architecture*, Feb. 1997.

[11] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. SPINE: A safe programmble and integrated network environment. In *Eighth ACM SIGOPS European Workshop*, Sept. 1998.

[12] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, May 1998.

[13] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, and P. Athanas. Implementing an API for distributed adaptive computing systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 222–230, April 1999.

[14] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *International Conference on Computer Architecture*, pages 260–267, Apr. 1994.

[15] M.-C. Roşu, K. Schwan, and R. Fujimoto. Supporting parallel applications on clusters of workstations: The intelligent network interface approach. In *Proceeding of the 6th International Symposium on High Performance Distributed Computing (HPDC 97)*, 1997.

[16] S. Sumimoto, H. Tezuka, A. Hori, H. Harada, T. Takahashi, and Y. Ishikawa. The design and evaluation of high performance communication using a Gigabit Ethernet. In *International Conference on Supercomputing*, pages 260–267, June 1999.

[17] K. D. Underwood, R. R. Sass, and W. B. Ligon, III. Cost effectiveness of an adaptable computing cluster. In *(to appear in) Proceedings of SC Conference*, Nov. 2001.

[18] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 40–53, Dec. 1995.

[19] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.