Exploiting Active CMP-based Devices in System Area Networks

Ming Hao
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853
haom@csl.cornell.edu

Mark Heinrich
School of Computer Science
University of Central Florida
Orlando, FL 32816
heinrich@cs.ucf.edu

Abstract

We explore the possibility of exploiting the computing power of chip-multiprocessor-based network interface controllers and switches in a system area network environment. We use stream processing applications as case studies to show that combining the compute-power of the host processor and a CMP-based NIC with 4 CPU cores can achieve speedup up to 1.30 even with the NIC running at one tenth the speed of the host processor. We also emphasize that a workload balance between the host CPU and the NIC CPUs is the key to achieving maximum speedup. With detailed simulation, we explore the effects of different NIC memory hierarchies on offloaded application performance. In addition, we propose the idea of two-level active I/O systems with both CMP-based NICs and CMP-based switches, and demonstrate that they can achieve even better performance than a system with active NICs or active switches alone.

1 Introduction

Many network interfaces for System Area Networks (SANs) use embedded processors to process a portion of the message passing protocol. This approach not only avoids the complexity of ASIC design and shortens the time-to-market, but also offers the flexibility of protocol debugging and upgrading. The availability of these intelligent NICs has motivated many researches to offload a myriad of functions to the NICs to boost system performance. Typically these functions fall in the category of processing small packets on the NIC to reduce interrupt and DMA setup overhead associated with transferring the packets over the PCI bus, or forwarding input packets directly to video display devices or the network output link to avoid the bandwidth bottleneck of the PCI bus.

However, the embedded processor on the NIC has much lower performance than the host processor because of power budget, cost, and other design constraints. Consequently, the functions appropriate for offloading are traditionally limited to simple packet pre-processing or checksum calculations. In addition, care must be taken to avoid any adverse effect on normal message protocol execution. In the past few years, several new SAN standards have been proposed, like PCI Express [21] and InfiniBand [8]. These new technologies significantly reduce the I/O bandwidth bottleneck (though avoiding unnecessary data transfer is still desirable). Further, these new system area networks enable tighter coupling of the network interface with the memory controller, and the overhead related to transferring packets between host memory and NIC memory is correspondingly reduced.

But high-level communication protocols will still consume many host processor cycles. With a 1 Gb/s duplex Ethernet link and Alacritech's 1000x1 TCP offload engine, 52.5% of two Pentium III 1 GHz processors are still required to move 1.75 Gb/s of data [9]. Thus, offloading code to the network interface to reduce host CPU utilization will continue to be desirable and important. With the advance of VLSI technology, integrating more than one CPU core in a NIC or switch is becoming feasible and can enable this higher-level offloading. For example, the Elan network interface [18] has a separate processor called the thread processor for higher-level communication protocols in addition to a *u*code processor that is responsible for low-level message passing. Also, some researchers have already tried to use network processors with multiple CPU cores to build network interfaces. Though network processors were originally designed for packet header processing within backbone routers, their popularity is rapidly driving down their cost. The Intel IXP1200 now only costs around $45 while integrating 6 microengines running at 200 MHz. With these CMP processors at the core of system area network interfaces, there is more computational power than needed to simply process low-level message passing protocols, suggesting the potential for enabling more aggressive offloading.

On the SAN switch side, there is also an opportunity for integrating multiple CPU cores. A current trend is to put network processors within switches to implement protocol conversion between the many different networking standards (e.g. Fibre Channel, iSCSI, InfiniBand, Ethernet, etc.). As a natural extension, we propose that application code can be offloaded to switches and cre-

ate active SAN switches. Running high-level application code enables the switch to filter out unnecessary data and reduce not only host CPU utilization but also the traffic through the SAN. One possible application is file copy from one disk to another. An active switch can redirect data so that it does not pass through the host at all.

In summary, we believe future intelligent network devices including NICs and switches will be CMP-based and thus provide many more opportunities for application offloading. This paper shows the case of stream processing in a database system that offloads its operators to take advantage of CMP-based devices to boost overall system performance. We focus on partitioning the applications across the host processors and the CMPs on the devices to maximize performance. Also, we show that combining active NICs and active switches will lead to better performance than either alone. We also propose new architecture support that makes programming these devices easier. In particular, we study the memory hierarchy of active NICs and show that unlike message header processing, caches are required for the embedded processors to achieve good performance for stream processing operations.

We describe our active NIC and switch microarchitecture in Section 2 and introduce our simulator and stream processing database system in Section 3. We discuss the performance benefits of exploiting active NICs and switches and examine detailed cache design trade-offs for active devices in Section 4 and Section 5. We describe related work in Section 6, and conclude in Section 7.

2 Architecture and Programming Model

The systems we consider in this paper can be abstracted as a group or cluster of compute nodes and storage devices connected by a switch-based SAN such as InfiniBand or PCI Express (formerly 3GIO) (the exact choice of switch-based system area network is not important). Figure 1 shows an example of such a SAN-based cluster. Compute nodes connect to the network via CMP-based active NICs, to which users can offload application code. We also propose replacing conventional switches with active switches which, in addition to routing messages from source to destination, can execute application-level code that may process messages more efficiently, save network or I/O bandwidth, reduce host processor utilization, or reduce application execution time [7].

Figure 2 shows the architecture of a CMP-based active NIC. There are multiple CPU cores on the NIC, each with a private instruction and data cache. We use one CPU, the protocol CPU, for message protocol processing and the others for offloading user applications. The assisting hardware includes checksum computing units and configuration and status registers that can only be accessed by the protocol CPU. SRAM is used as backing store for the caches, packet buffering, and application code and data storage. CPU core accesses to buffered data packets are uncached so that cache coherence between the embedded CPU cores is unnecessary. To hide as many of the hardware details as possible from the programmer while keeping a familiar programming paradigm, messages that will be processed by NIC CPU cores are memory-mapped into a contiguous unused area of physical memory. Thus, offloaded applications can use normal load and store instructions to access the message payload, similar to a memory-mapped file. There is special hardware called an ATB (address translation buffer) that acts like a TLB and implements these CPU-to-message-payload-buffer address translations.

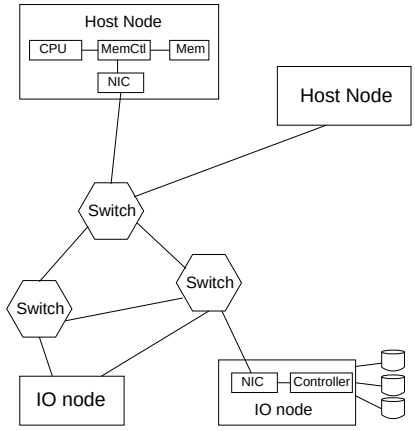

Figure 1. A typical SAN-based cluster

The protocol CPU runs the normal message passing firmware. In either a special on-chip buffer or SRAM, it maintains a table (JumpTable) of starting PCs for offloaded applications stored in either the on-chip SRAM or in external memory, and is responsible for invoking an offloaded application on a CPU core and updating its ATB with the addresses of its incoming packets. To invoke a handler on a CPU core, the protocol CPU writes the starting PC and message header into a special per-CPU queue on which the CPU cores poll when idle. The scratchpad is a small high-speed SRAM that provides fast communication and synchronization among CPU cores.

Figure 3 shows the architecture of a CMP-based active switch with 8 ports. Solid and dashed lines signify data and control lines, respectively. The shaded portion of the figure is a normal switch based on a central output queue scheme similar to that in the IBM Switch-3 [20]. The key active components are multiple switch processing cores (SPs), their cache structures, and on-chip data buffers (DB) that play an integral role in the processing of incoming and composing of outgoing messages. The data buffers are the data interface between the active and non-active portions of the switch and are connected

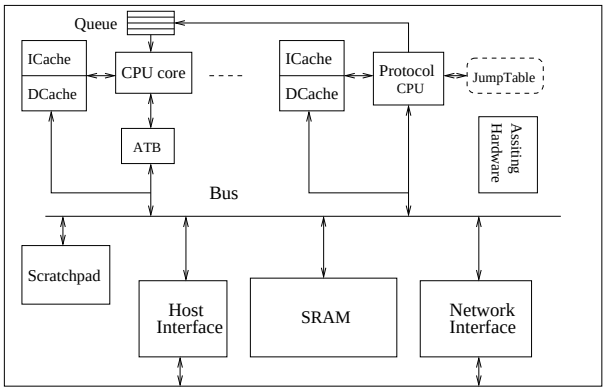

Figure 2. CMP-based active NIC

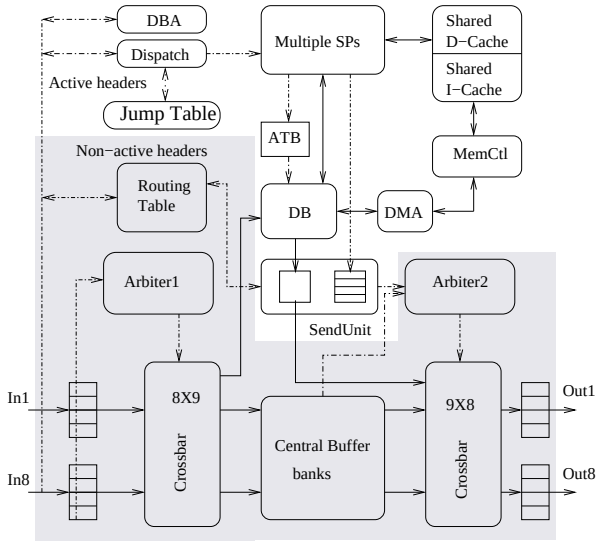

Figure 3. Switch architecture with multiple SPs

to the Crossbar. The ATB here is the same as the one in the NIC. Detailed information on the micro-architecture can be found in [7].

We adopt a stream-based programming model for both active NICs and switches. We call the application code running on the embedded CPU cores *handlers*. Handlers are invoked by incoming active messages in the style of message-driven processors [12, 15, 22]. Handler dispatch is based on information in the 128-bit header of the message. As mentioned above, with the help of the ATB, handlers can access the message payload like a memory-mapped file, greatly easing the access of data structures that cross message packet boundaries. A detailed discussion of the programming model can also be found in [7]. For both switches and NICs, we allow handlers to freely allocate memory. For protection reasons, we assume for now that only a trusted system programmer can offload applications to active devices.

3 Methodology

In this section we briefly discuss the simulation environment and architectural parameters we use to evaluate our active NIC and active switch systems. Our simulator models a MIPS-based host processor, its cache subsystem, memory controller, and memory system in detail based on a simulator that was validated against real hardware in [10]. Our NIC model is based on the Myrinet network interface card [14], enhanced with multiple CPU cores. It has 3 DMA engines, two for transporting data between the NIC's SRAM and network interface and one for the host interface. The simulator accurately models the message protocol execution latency and DMA setup and interrupt overhead. The NIC exports a queue-pair communication interface to the host. The embedded processors of both NICs and switches are modeled based on an R4000 with 5 pipeline stages. Embedded cores run at one tenth the speed of host processors. We also model a CMP-based SAN switch connected to the memory controller via links and the NIC (as shown in Figure 1) with embedded processors (as shown in Figure 3). Switches route packets in a virtual cut-through style. Its detailed design can be found in [7].

Our host processor runs at 2 GHz and is equipped with separate 32 KB primary instruction and data caches, both of which are two-way set associative. The secondary cache is unified, 512 KB, two-way set associative, and has a line size of 128 bytes. Our simulator accurately models an RDRAM memory system for both the host and switch. The maximum bandwidth of both systems is 1.6 GB/s. The latency of a page hit is 100ns and 122ns for a page miss. Details can be found in [4]. Active NICs use SRAM as backing store. As in the Myrinet NIC, the SRAM is pipelined and has an access latency of 3 cycles. The embedded processors on both NICs and switches run at 200 MHz with an 8 KB, two-way set-associative instruction cache with 64-byte lines and a 8 KB two-way set-associative data cache with 32-byte lines.

The main parameters are summarized in Table 1.

Table 1. Architectural Simulation Parameters

Host	CPU: 2 GHz, D$: 32KB I$: 32KB, L2$: 512KB
NIC	CPU: 200 MHz, D$: 8KB, I$: 8KB SRAM: 4MB, Latency: 3cycles
Switch	CPU: 200 MHz, D$: 8KB, I$: 8KB Data Buffers: 32, Latency: 100ns
Memory	120ns, 1.6GB/s
Network	1 Gb/s

Our evaluation has three main goals. The first is to demonstrate the potential benefits of higher-level offloading of application code to a CMP-based NIC. The second

is to study the effects of various architectural parameters of CMP-based devices on the performance of offloaded applications. We focus on the memory hierarchy of active NICs in this paper because it is the most important determinant of offloaded application performance. The third goal is to show the benefits of combining CMP-based devices together (both NICs and switches) to form a multi-level active I/O system.

Our main application is a stream processing database system. This application domain is becoming increasingly important and includes applications like sensor data processing. Data stream processing is typically implemented by the execution of queries over input streams. Those queries are comprised of many basic query operators that can run in parallel—either in a pipelined fashion, within a stream, or across different streams. This modularity and coarse-grain data parallelism fits a CMP architecture well. The specific database system we choose is called Aurora [1], a research system that manages data streams for monitoring applications. In Aurora, input streams consist of a series of tuples that have the form $(ts, v1, ..., vn)$. The system adds ts, a timestamp, to each tuple. The other fields are application-specific. Aurora uses primitive operators to construct queries. These are analogous to operators in a relational algebra; however, they differ in fundamental ways to address the special requirements of stream processing. Table 2 summarizes the Aurora operators that we use in our evaluation.

Table 2. Operators and Their Forms

Operators	Forms
`Map`	(B1 = F1,...,Bm = Fm)(S)
`Filter`	(P1,...,Pm)(S)
`BSort`	(Assuming O)(S)
`Aggregate`	(F, Assuming O, Size s, Advance i) (S)
`Join`	(P, Size s, Left Assuming O1, Right Assuming O2) (S1, S2)
`Resample`	Resample (F, Size s, Left Assuming O1, Right Assuming O2) (S1, S2)

All the operators have the same basic meaning as in normal database systems. Readers are referred to [1] for detailed descriptions of each operator. Here we only give a brief explanation on the function of each operation. `Map` is an extended projection operation; `Filter` is used to select tuples satisfying specified predicates. The remaining 4 operators are called *order-sensitive* since they can only be guaranteed to execute with finite buffer space and in finite time if they can assume some ordering (potentially with bounded disorder) over their input streams. Order specifications have the form

$$Order(On\ A, Slack\ n, GroupBy\ B1, ..., Bm)$$

where A is the ordering attribute and Slack n is the number of out-of-order tuples considered by the operators. Specifically, `BSort` performs a buffer-based approximate sort equivalent to n passes of a bubble sort where $slack = n$; `Aggregate` performs aggregation within sliding windows over input streams; `Join` is a join operation over two data streams; `Resample` is an asymmetric, semijoin-like synchronization operator that can be used to align pairs of streams.

In addition to the above stream processing system, we also use the `MD5` and `Blowfish` decryption algorithms to particularly show that multiple CPU cores on an active NIC can work on a single stream by partitioning the input stream among them. Except `Filter` and `Map`, it is not easy to do this for stream processing in Aurora since other operators are *order-sensitive*. The original `MD5` uses block chaining to ensure sensitivity to block order that prevents arbitrary parallelism. We slightly alter `MD5` to run on multiple CPUs. There should be a predetermined finite number of blocks processed from independent seeds, such that the I^{th} block is part of the "I mod K"th chain. The resulting K digests themselves form a message, which can be MD5-encoded using a single-block algorithm. Based on this algorithm, each CPU core is assigned a packet in an interleaved fashion and sends a partial digest to the host independently. `Blowfish` is a symmetric block cipher algorithm that encrypts and decrypts data in blocks of 64 bits. It makes use of a key that ranges from 32 bits to 448 bits. Although CBC (Cipher Block Chaining) mode requires each plaintext block to be XORed with the previous ciphertext block, this dependence does not affect executing the decrypt in parallel. We run block decrypt in parallel by assigning data packets to each CPU core in an interleaved way as in `MD5`. Sending decrypted output is ensured to keep original packet order.

4 CMP-based NIC Results

CPU cores on a NIC can either work together with the host CPU in a pipelined fashion, or work independently on unrelated streams in parallel. In this section, we will show that significant performance improvement can be achieved in all these cases.

First, to chain NIC CPU cores together with the host CPU in a pipelined fashion, we need to partition a query into a host portion and a NIC portion. We use a synthetic query as the benchmark that consists of two operators: a `Filter` that runs on the NIC and a sliding-window based `Aggregate` operator that runs on the host. The tuple has a size of 164 bytes and streams into the NIC continuously from the network. Since the query has only two operators, we run the more complicated `Aggregate` operator on the host and `Filter` on the NIC. There is only one input data stream. When we use more than one CPU core on the NIC, the input stream is partitioned in a round-robin fashion among the cores.

Figure 4 shows the active NIC speedup relative to the

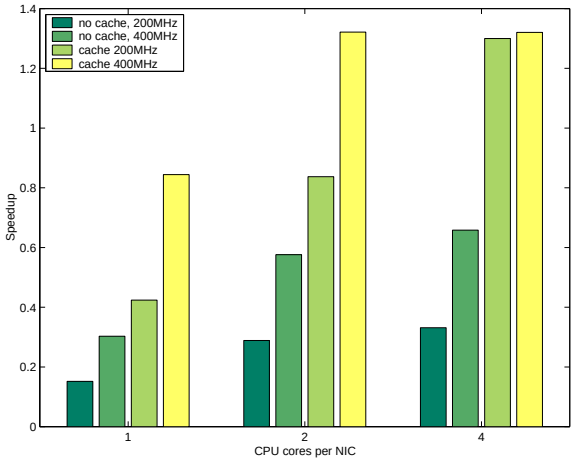

Figure 4. Speedup of Partitioned Query with Filter and Aggregate

case without using any NIC CPU cores at all (running both operators on the host). We show cases with NIC CPU cores running at one-tenth (200 MHz) and one-fifth (400 MHz) the host processor speed. We consider two NIC memory configurations. One is with both an 8 KB instruction cache and an 8 KB data cache with SRAM as backing store, while the other uses only SRAM. In both cases, streaming data is uncached. With one or two 200 MHz CPUs cores, the slower NIC CPUs are a bottleneck and the host is under-utilized. As a result, execution time actually increases relative to running both operators on the host. A balanced workload is reached with 4 200 MHz CPU cores plus caches, where the active NIC achieves a speedup of 1.30. When the NIC CPU frequency is doubled, the balance is reached with only 2 CPU cores plus caches, with no improvement at 4 400 MHz cores because the NIC's computing power becomes under-utilized and the host becomes the bottleneck. The NIC CPU cores form a two-stage pipeline with the host processor, and the maximum speedup can only be achieved when the application partitioning balances these two stages. The computational intensity of each operator strongly depends not only on the query itself and the values in the input streams but also on the number of available NIC CPU cores. Therefore, an ideal scheme is one that dynamically decides whether to offload operators and which to offload based on this information. As an experiment, we adjusted the parameters of the same `Filter` and `Aggregate` combination so that 4 400MHz CPU cores can reach a load balance with the host and achieve a speedup of 1.76. Figure 4 also indicates the importance of both the instruction and data caches. With only SRAM, the NIC still can not keep up with the processing speed of the host even with 4 400 MHz CPUs.

Figure 5 shows the host CPU utilization in the above simulations. Host utilization is defined as $1 - IdleTime/TotalTime$. We can clearly see that the host is under-utilized when the NIC is the bottleneck, thus performance suffers, though saved host CPU cycles can be used for other queries in a multiprocessing environment.

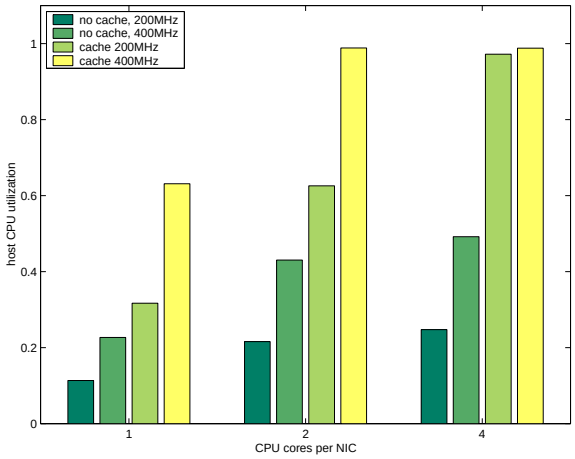

Figure 5. Host Utilization of Partitioned Query

In the above example, the CPUs of the NIC work together on the same input steam. The key to this execution model is computation balance and synchronization among participating CPU cores. Though `Filter` can run in this way, non-uniform selectivity can cause load imbalance. `MD5` and `Blowfish` are two typical applications that better fit this model: both apply the same operations repeatedly on small units of data. Figure 6 shows the speedup of `MD5` and `Blowfish` decryption with different numbers of cores. The host CPU remains idle for this measurement. In the case of `MD5`, CMP-based NICs achieve a speedup of 1.98 and 3.92 for 2 and 4 CPUs, respectively, because there is almost no synchronization overhead. `Blowfish` decryption achieves a speedup of 3.45 for 4 cores, slightly worse than `MD5` because of the need to synchronize to ensure correct output packet order. We want to point out that most network processors also use this processing model, but at the granularity of packets. Since we map input packets into a contiguous address space, the NIC CPUs can partition data in a more flexible manner.

Obviously, in addition to partitioning a single data stream across multiple CPU cores, each CPU core can also be used to process different streams. This is particularly useful for order-sensitive operators where it is relatively difficult to partition the data streams. Next, we will compare the processing throughput of 6 operators, respectively. They are `Join`, `Resampling`, `Aggregate`, `Map`, `Filter` and `BSort`. We use the same stream as in the above experiments, with a tuple size of 164 bytes. Here we briefly introduce what each operator does. `Filter` selects 20% of tuples by checking an integer field; `Map` selects three integer fields from original tuples; `Aggregate`

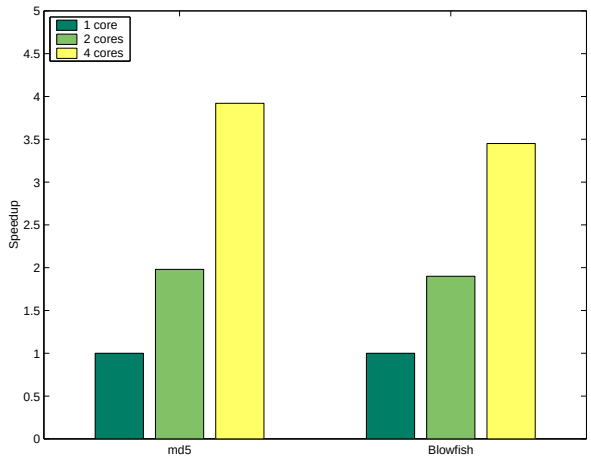

Figure 6. Speedup of MD5 and Blowfish Decryption

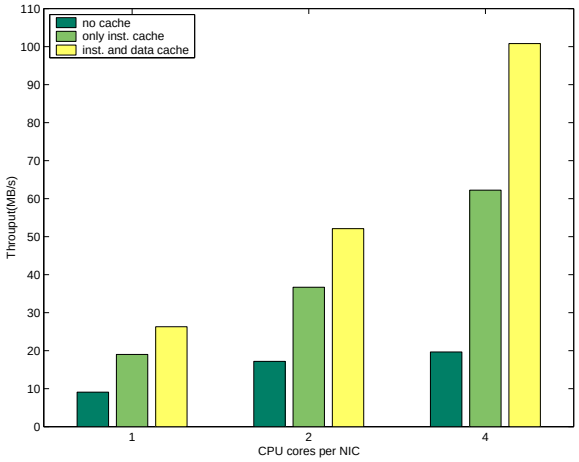

Figure 7. Throughput of Filter Operator

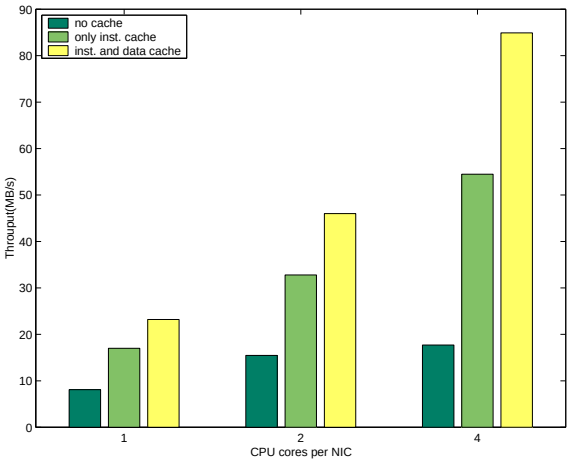

Figure 8. Throughput of Map Operator

performs a sliding window-based average operation on one integer field. `BSort` sorts input tuples with a window of 20 tuples; `Resampling` selects 10% of tuples by resampling based on an average operation; `Join` performs a window-based join operation with a window size of 10 tuples. For each operator, we also vary the number of processors. In all cases, each NIC CPU core and the host CPU run the same operator but process independent input data streams. The metric we use is the aggregate processing throughput of all participating CPUs.

We particularly focus on the performance impact of the NIC memory subsystem. As previously mentioned, using CMP-based network processors as kernel components of programmable NICs will become increasingly cost-effective as their variety of usage widens. Since the major data references of network processors are either the streaming data (the message payload) that have low temporal and spatial locality, data caches are typically not very effective in comparison to an SRAM buffer or multiple memory interface units. But with offloaded applications running on the NIC, the situation is different. In our simulations with data stream operators, we find that caches are quite effective at buffering the most frequently used instructions and application data other than the streaming data (though the specific working set size strongly depends on query and input data characteristics). In the experiments, we consider 3 memory hierarchies: SRAM only, a private instruction cache and SRAM, and both a private instruction cache and data cache plus SRAM. Streaming data accesses remain uncached because of their poor locality. We consider only private caches here. We did not consider a shared cache structure, though it would allow the sharing of data/code between NIC processor cores, because of the necessity of multi-porting the shared cache (for performance) and because of the resulting implementation complexity and potential impact on NIC core clock rate.

Figure 7 and Figure 8 show the aggregate throughput of `Filter` and `Map`, respectively. Since these two are relatively simple operators and using the host CPU only would make the network input link a bottleneck, we only show the aggregate bandwidth of each operator across different numbers of NIC CPU cores. With 1 Gb/s input link bandwidth, `Filter` and `Map` achieve an aggregate throughput of 106.1 MB/s and 84.9 MB/s, respectively, with 4 CPU cores plus both instruction and data caches. The 8 KB caches are large enough for their instruction and data working sets. In contrast, with 4 CPU cores, `Filter` and `Map` achieve only 19.7 MB/s and 17.7 MB/s, respectively, in the cases without caches. Adding only an instruction cache greatly improves the throughput since CPU cores need to fetch instructions from SRAM each cycle. Still, adding a data cache helps even further. For `Filter` with 4 CPU cores, the throughput achieved with a data cache is 1.62 times larger than without one. Next, for all the rest of operators, we show their aggregate bandwidth along with that of the host CPU alone. Zero NIC CPU cores means the throughput achieved with only the host CPU.

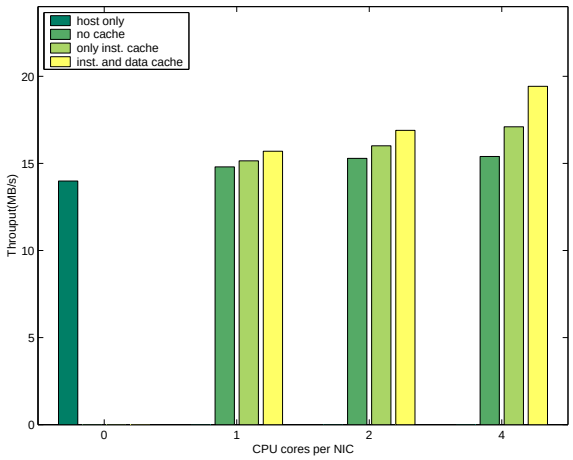

Figure 9. Throughput of Aggregate Operator

For the `Aggregate` operator, we can see from Figure 9 that the achieved throughput is much lower than either the `Filter` or `Map` operators. `Aggregate` is a relatively compute-intensive operator. With only the host CPU, it achieves 14.0 MB/s. Adding NIC CPU cores significantly improves the aggregate throughput. 4 NIC CPU cores with caches achieves 1.39 times higher throughput than the case with only the host CPU. 8 KB caches can cover its working sets and it only spends 8.0% and 2.0% of its execution time on instruction cache misses and data cache misses, respectively. Caches greatly improve the processing throughput. In the case of 4 CPU cores with caches, it achieves 1.26 times higher throughput than without any caches.

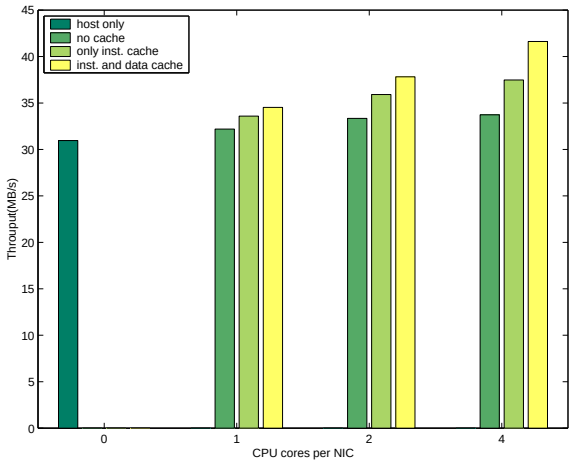

Figure 10. Throughput of Resample Operator

The throughput of the `Resample` operator reflects a similar trend as `Aggregate`. With only the host, it achieves a throughput of 31.0 MB/s. Using NIC CPU cores clearly has a large benefit. Considering the cases with caches, 1.22 times and 1.34 times more throughput are achieved with 2 CPU cores and 4 CPU cores, respectively. The improvement is smaller in comparison to the `Aggregate` operator because the `Resample` operator has an instruction working set between 16 KB and 32 KB. It spends 24.3% of its execution time on instruction cache stall. Similarly, caches improve the throughput achieved by NIC CPU cores. In the case of 4 CPU cores with caches, it achieves 1.10 and 1.23 times higher throughput than the case with only an instruction cache and the case without caches, respectively.

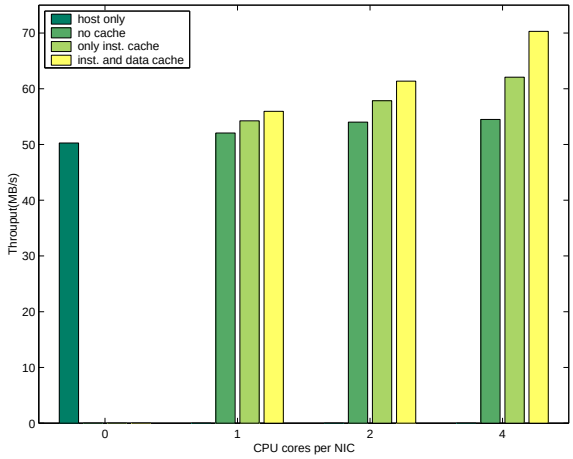

Figure 11. Throughput of Join Operator

Figure 11 shows the performance of the `Join` operator. With only a host CPU, it achieves a throughput of 50.3 MB/s. Like the `Aggregate` and `Resample` operators, adding NIC CPU cores results in a significant throughput increase. Considering cases of 4 CPU cores with both an instruction cache and a data cache, active NIC systems can achieve a throughput 1.4 times higher than the case with only a host CPU. This large improvement also results from the fact that the host spends 18.8% of its execution time on cache misses. Further, 8 KB caches are small for this specific `Join` operation, which spends 15.6% of its execution time on cache stall. Still, with 4 CPU cores, the throughput of the case using caches is 1.29 times higher than the case without any caches.

Figure 12 shows the aggregate throughput of the `BSort` operator under different cases. With only a host CPU, it achieves a throughput of 93.0 MB/s. Adding NIC CPU cores does not increase the achieved throughput much because the network input link soon becomes the bottleneck and the NIC CPU cores are under-utilized. With 4 CPU cores on a NIC with both instruction and data caches, it achieves only 1.21 times the throughput of the case with only a host CPU. As an experiment we increased the network link bandwidth to 10Gb/s for Bsort and then indeed having 4 NIC CPUs with both instruction cache and data cache improved the throughput by a factor of 1.44 over the host alone.

We also want to point out that though all operators have small working sets, some have a relatively large in-

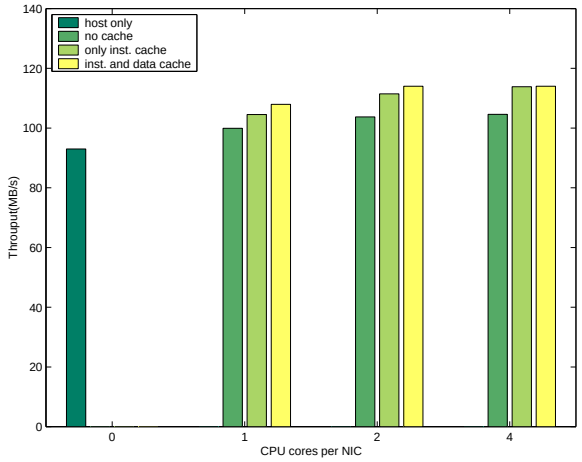

Figure 12. Throughput of BSort Operator

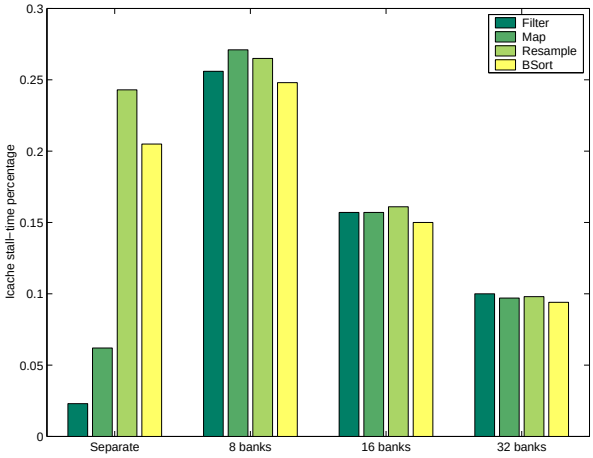

Figure 13. Instruction Cache Performance

struction footprint and 8 KB NIC instruction caches are not sufficient. Both `Resample` and `Bsort` operators spend over 20% of execution time on instruction cache stall. Current network processors provide only a very small instruction cache or on-chip SRAM since their limited functionality results in a small code size. One possible solution to this problem is to use a shared instruction cache to increase utilization. If all cores run the same handler at the same time, obviously a shared cache is beneficial. But shared caches can also introduce destructive interference between different handlers. Figure 13 shows the performance of both private and multi-banked shared instruction caches. We consider three different numbers of banks for the shared cache configurations. The private instruction cache is 2-way set-associative, while the shared cache is 8-way set-associative. For the shared cache, two requests can proceed in parallel to different banks, but not to the same bank. The total cache size, 32 KB, is the same for all four configurations. Four different operators are executed simultaneously and the figure shows the percentage of instruction stall-time experienced by each operator. With a private instruction cache, we can see that both `Fill` and `Map` have negligible instruction cache stall time with an 8 KB cache. But with a shared instruction cache, their instruction stall-time percentage increases to the same level as `BSort` and `Resample` because of destructive interference. Since the instruction cache needs to be accessed almost every cycle, a shared instruction cache usually exhibits larger interference than a shared data cache. We also see that the number of banks clearly determines the probability of conflicts and thus the percentage of stall time. To avoid interference from operators with a large instruction footprint, a hybrid shared cache scheme may be used with the combination of a small private cache for each core and a relatively large shared cache, though this would increase the complexity of the cache subsystem. We expect that limited instruction cache sizes will limit the complexity of user functions that can be executed on an active NIC.

In summary, using CPU cores on the NIC can significantly boost the performance of operations in the stream processing system we consider here. These CPU cores can be used either to form a pipeline with the host CPU or to work on independent data streams. Further, both instruction caches and data caches are necessary to achieve good performance for offloaded operators, even with an SRAM backing store.

5 Two-Level Active I/O Systems

In [7], we proposed the idea of active switches and demonstrated its potential performance benefits. In comparison to active NICs, active switches are closer to the data source and thus can reduce the network workload between host systems and switches. Further, for those active switches that connect two different networks, avoiding converting data from a simpler protocol to a more complicated one may lead to extra benefits. For example, Fibre Channel is currently widely-used in storage area networks. But a new standard, iSCSI, is gaining attention because of thee ubiquitous usage of Ethernet. Switches with embedded processors can convert traffic between these two protocols and merge the two networks at the switch. In comparison to the Fibre Channel protocol, the IP stack is more compute-intensive. Processing Fibre channel data right at the switch can help reduce the data traffic that must go over Ethernet.

We believe active NICs and active switches can be combined together to achieve better performance than either of them alone. The following example shows a simple filter running on either an active CMP-based NIC alone or an active CMP-based switch alone or both together. The filter checks one integer field in the input record and filters out those whose value is smaller than an integer constant. The selectivity is 0.8. The result is shown in Figure 14. The number in the labels on the

x-axis stands for the number of CPU cores. The hybrid case means that half of the CPUs are running on the NIC and the other half on the switch. Since the network link has only 1 Gb/s bandwidth, with 4 CPUs on the NIC the overall throughput achieved is limited by the input link bandwidth. However, utilizing two CPUs on the switch in conjunction with 2 CPUs on the NIC can avoid this bottleneck and achieve 181.1 MB/s—more than 50% higher than the case with same number of CPUs on the NIC alone.

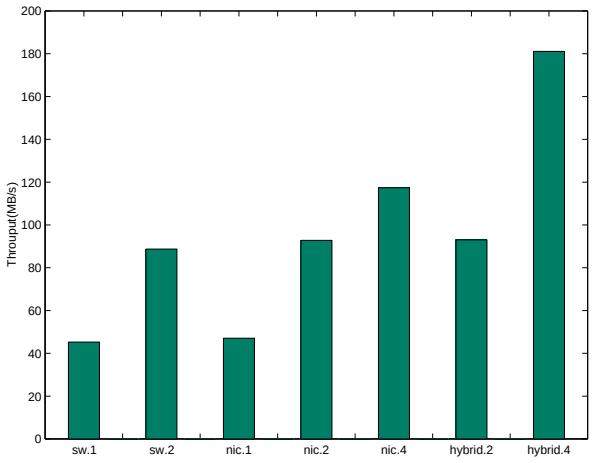

Figure 14. Throughput of Filter

6 Related Work

There is a large body of research attempting to exploit intelligent NICs. These efforts are mainly in two directions. First, there is a desire to avoid communication overhead between the host memory and the NIC via the PCI bus, particularly for small packets since they have relatively high DMA and interrupt overhead. The computation performed on these NICs is limited to simple operations so as to avoid overloading the slow NIC CPU, which also needs to execute the message passing protocol. [13] offloaded applications that require small amounts of computation on a large number of small packets. The amount of computation ranges from 0.75 to 2 instructions/byte. One example is an airline global information system that receives a continuous stream of small messages from all airports, tickets agents, baggage handlers and the FAA. [2] does NIC-based MPI collective reductions (supporting simple integer and floating point reductions like binary AND, OR, MAX, MIN and SUM operations). Though the NIC's processor is much slower than the host processor, the overhead reduction from eliminating messages between the host and the NIC makes this scheme achieve up to a factor 1.19 performance improvement. [3] experimented with offloading part of a network traffic analysis code to the NIC but claimed that the slow CPU cores on their NIC could not do any useful work besides message protocol processing. Another interesting example is [16], which uses programmable NICs to improve the performance of a Time-Warp application. One of the optimizations proposed in the paper is to migrate GVT computation to the NIC since this computation is not on the critical path of the Time-Warp simulation and can be performed in the background. It is also not computationally intensive and does not place excessive load on the NIC. I/O bandwidth is also saved by intercepting and preventing messages from going across the I/O bus.

The second research direction is avoiding the PCI bandwidth bottleneck. Until recently, the PCI bus has been a bottleneck for applications that communicate large amounts of data like forwarding servers and media servers. This motivates the idea of decoding video streams on the NIC and passing data directly to a display system or caching data on the NIC and forwarding it directly from there to bypass the PCI bus. Along this direction, [5] is a typical example that provides a safe, programmable, and integrated network environment. It enables applications to compute directly on the network interface. This environment allows network-oriented applications to communicate with other applications executing on the host CPU, peer devices, and remote nodes with low latency and high efficiency. [23] proposes a payload caching technique that extends the network interface to cache portions of the incoming packet stream, enabling the system to forward data directly from the cache. [11] applies the idea of caching on the NIC to webservers. Its performance gains mainly come from avoiding the PCI bus bandwidth bottleneck.

Our approach is different from the research efforts above in that it assumes a CMP-based NIC and exploits the computing power on a NIC while simultaneously reducing data traffic into host memory. It also first proposes the idea of two-level active device systems by combining both active NICs and active switches together to provide more computing power and reducing the data traffic flowing through the network.

7 Conclusions

Emerging I/O standards like InfiniBand and PCI Express eliminate the bandwidth bottleneck of the PCI bus and reduce message transfer overhead between host memory and the NIC because of the corresponding tighter coupling of the NIC and the memory controller. Offloading applications to the NIC will focus on exploiting the available computing power on a CMP-based network interface controller to reduce the host CPU's utilization and improve application performance. In this paper, we use stream processing applications as case studies, showing that combining the computing power of the host and the NIC can achieve a speedup of 1.31 with 4 NIC CPU cores running at one tenth the speed of the host CPU.

We also point out that a workload balance between the host CPU and the NIC CPU cores is the key to achieving maximum speedup.

The decision whether to process messages on the NIC and what kind of applications to offload depends strongly not only on the performance difference between the NIC CPUs and the host CPU, but also on input data and application characteristics. Furthermore, each NIC CPU core can process independent data streams and achieve near linear aggregate throughput improvement. Via detailed simulation, we also show that private caches for each NIC CPU core are necessary to achieve good performance.

In addition to active NICs, active switches equipped with multiple CPU cores can also execute offloaded user applications. Further, we propose the idea of combining active NICs and active switches together in the same system and demonstrate that such a system can not only provide more computing power, but can also reduce traffic over the input links of NICs, thus achieving better performance than either active NICs or active switches alone. Specifically, 50% more throughput can be achieved in a two-level active device system than using only a CMP-based NIC design.

Acknowledgments

This research was supported by US National Science Foundation CAREER Award CCR-0340600.

References

[1] D. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. In *VLDB Journal*, Vol. 12, No. 2, August, 2003.

[2] D. Buntinas and D. K. Panda. NIC-Based Reduction inMyrinet Clusters: Is It Beneficial? In *The Second Workshop on Novel Uses of System Area Networks*, Feburary 2003.

[3] C. Cranor et al. Gigascope: A Stream Database for Network Applications. In *SIGMOD (Industrial Track)*, June 2003.

[4] Direct RDRAM 256/288-Mbit Specification. http://www.rdram.com/documentation/.

[5] M. E. Fiuczynski et al. SPINE - A Safe Programmable and Integrated Network Environment. In*Proceedings of the Eighth ACM SIGOPS European Workshop*, 1998.

[6] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, November 2000.

[7] M. Hao and M. Heinrich. Active I/O Switches in System Area Networks. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 365–376, February 2003.

[8] InfiniBand Architecture Specification, Volume 1.0, Release 1.0. InfiniBand Trade Association, October 24, 2000.

[9] InfiniBand and TCP in the Data Center. http://www.mellanox.com/technology

[10] D. Kim, M. Chaudhuri and M. Heinrich. Leveraging Cache Coherence in Active Memory Systems. In *Proceedings of the 16th ACM International Conference on Supercomputing*, 2002.

[11] H. Kim, V. S. Pai, and S. Rixner. Increasing web server throughput with network interface data caching. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems* , October 2002.

[12] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[13] K. Mackenzie et al. A Intel IXP1200-based Network Interface. In *The Second Workshop on Novel Uses of System Area Networks*, Feburary 2003.

[14] Myrinet/PCI-X and Myrinet/PCI Host Interfaces. http://www.myrinet.com/myrinet/PCIX/m3f2-pcixe.html

[15] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224–235, May 1993.

[16] R. Noronha and N. B. Abu-Ghazaleh. Using Programmable NICs for Time-Warp Optimization. In *6th International Parallel and Distributed Processing Symposium*, April 2002.

[17] D. Patterson and J. Hennessey. Computer Architecture: A Quantitative Approach, Morgan Kaufman, 2nd edition, 1996.

[18] F. Petrini et al. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *A Symposium on High Performance Interconnects 13*, August 2001.

[19] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proceedings of the 24th international Conference on Very Large Databases (VLDB '98)*, August 1998.

[20] C. B. Stunkel et al. A new switch chip for IBM RS/6000 SP systems. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, January 1999.

[21] Third Generation I/O Architecture. http://www.intel.com/technology/3GIO/

[22] T. von Eicken et al. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[23] K. Yocum and J. Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *2001 USENIX Technical Conference*, June 2001.