Active I/O Switches in System Area Networks

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

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

Abstract

We present an active switch architecture to improve the performance of systems connected via system area networks. Our programmable active switches not only flexibly route packets between any combination of hosts and I/O devices, but also have the capability of running application-level code, forming a parallel processor in the SAN subsystem. By replacing existing SAN-based switches with a new active switch architecture, we can design a prototype system with otherwise commercially available, commodity parts that can dramatically speed up data-intensive applications and workloads on modern multi-programmed servers. We explain the programming model and detail the microarchitecture of our active switch, and analyze simulation results for nine benchmark applications that highlight various advantages of active switch-based systems.

1 Introduction

In this paper, we examine the effect of incorporating *intelligent* or *active* switches into an otherwise standard cluster connected via a system-area network (SAN). SANs are becoming increasingly popular in servers and systems with large I/O requirements because the SAN network interface is being integrated into commodity memory controllers, reducing the latency to and from I/O devices, and increasing the bandwidth over traditional I/O solutions. Though there has been work on programmable intelligent or active I/O devices, we will show that there are several advantages (both technical and economical) to active switch-based systems, and in fact the use of active switches is orthogonal to the use of intelligent I/O devices.

The systems we consider in this paper can be abstracted as a group or cluster of compute nodes and storage devices connected by a switched-based system-area network such as InfiniBand [16] or PCI Express (formerly 3GIO) [24] (the exact choice of switch-based system area network is not important). Figure 1 shows an example of such a SAN-based cluster. HCA (host channel adapter) and TCA (target channel adapter) are terminology from the InfiniBand SAN architecture—other systems refer to

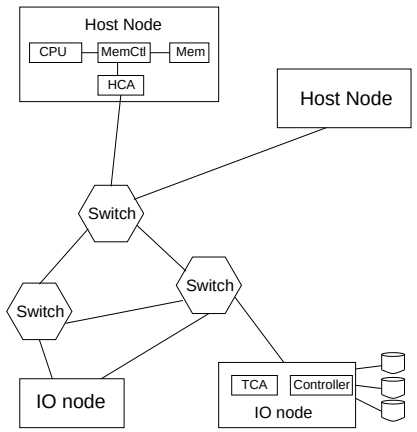

Figure 1. A typical SAN-based cluster

these pieces as the NI and NIC, respectively. All communication between end nodes, whether it is message passing between compute nodes or an I/O operation between a computing node and a storage device, must traverse one or more network switches. The central position of switches in the cluster gives them the potential to play an *active* role in the system rather than simply routing messages from source to destination. In particular, active switches may contain one or more embedded processors that 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. For the normal non-active system, only compute nodes are user-programmable. Storage and network devices can only be accessed through software interfaces exported by the operating system and the network switches are transparent to users. However, on active switch systems, we view both switches and host processors as user-programmable compute nodes, possibly of different capabilities. In this system, the host processors and active switches cooperate to finish tasks faster than either could do alone. The key is to decompose potential applications into tasks that are best suited for high-performance compute-nodes and tasks that are best suited for active switch nodes located in the network and closer to the I/O system. Of course, switches still need to implement their core switching functions for both active and non-active messages. We discuss more details of

the programming model in Section 2 and describe our active switch microarchitecture in Section 3. Several active switch applications and their performance are discussed in Section 5. We find that active switch systems can achieve speedup up to 5.9 on suitable applications, while simultaneously dramatically lowering host processor utilization and network bandwidth requirements.

2 Programming Model

Our active switch architecture has two main design goals. First, the presence of active switches should not degrade the performance of (the likely more common) non-active messages. Second, active switches should be easy to use, exporting a simple programming model to the user. We use a combination of hardware and software techniques to realize each of these goals. In Section 3, we focus on hardware and detail our active switch architecture. In this section, we outline the stream-based programming model for our active SAN switch. Section 2.1 describes the overall environment and philosophy of our model, while Section 2.2 focuses on the details of active switch programming via a short example.

2.1 Active Switch Environment

Active switches are best suited for streaming applications, applications with small working sets, or those that can be pipelined in some way. An active switch *handler*, the code running on the switch, accesses its input data in a memory-mapped fashion as discussed in detail below. The overall goal is to hide as many of the low-level hardware details as possible from the programmer, simplifying the task of writing switch handlers.

The key step needed to run applications on active I/O switches in system area networks is that the conventional program must be divided into two parts—one that runs on the host processor as normal, and the other that runs on the active switch. Incoming messages to the active switch invoke handlers in the style of message-driven processors [18, 19, 25], based on information in the 128-bit header of the message. When an active message has a switch as its destination, that switch extracts a handler ID field from the header and uses it to index into a table to get the corresponding program counter for that handler and invokes it on an available switch processor.

For protection reasons, we assume that there is a small run-time kernel for active switches that can initiate necessary I/O requests and allocate memory for handlers. Switch handlers are not allowed to allocate memory freely. This is not a severe limitation since, as we shall see, active switches typically process data from their on-chip data buffers and only a small portion of their data structures need to be allocated in memory. This embedded kernel concept is used in the active disk model [1], and researchers have recently begun work on splitting the operating system into portions that run on the host processor and portions that run on intelligent I/O devices or switches [8]. For most of the benchmarks in this paper, the I/O requests are initiated by the host processor and the handlers process the data as they flow through the switch (only our Tar benchmark initiates disk requests from an active switch handler). This requires only modest kernel support on the switch (less than in previous active disk proposals).

2.2 Handler programming

Active switch handlers have common characteristics that are determined by the switch architecture and location within the cluster. First, they must not be compute-intensive. This is obvious since the host processor is more powerful and has larger caches, making it more amenable to compute-intensive tasks. A less obvious reason is that more computation on the switch results in higher switch occupancies and lower network throughput. Second, handlers should have a relatively small working set, or preferably, process data in streams. This follows from our design decision that an active switch should process data from its on-chip buffers (or caches) as much as possible without copying data into its local memory and incurring extra memory overhead. Our results will show that a few on-chip buffers and a small 1 KB data cache are sufficient to achieve speedup on many applications. However, because of this limited on-chip buffering, active switch handlers perform better with stream-based or small working sets. In addition, the need for fewer active handler buffers leaves more buffers available to maintain non-active switch throughput, and the streaming model ensures that active buffers are released as soon as possible, simplifying buffer management.

Still, programming active switches can be challenging. As we mentioned above, our programming model for active switches attempts to hide as many of the hardware details as possible from the programmer while keeping a familiar programming paradigm. The key programming model problem is how to address the stream data in the on-chip buffers. Our solution is to memory-map the buffers into a contiguous physical memory area. Thus, handlers can use normal load and store instructions to access the buffers and do not need to know which data is stored in which buffer—they just need to know the physical memory address. The physical memory address is translated either into a `(bufId, offset)` pair or into a normal physical address and sent either to the data buffers or data cache, respectively. Our switch has an address translation buffer (ATB) that performs the address-to-`bufId` translation if that data is currently present in one of the 16 on-chip buffers.

The following pseudo-code shows the common structure of a handler and demonstrates how a handler can

seamlessly access both the arguments sent by the host and a file from a storage device via this memory-mapping mechanism.

```
Byte *buf=ADDRESS, *arg=ADDRESS2;
int i,off = 0,file_len,bufSz,MTU;
ReadArg(arg);
for(i=0; i<file_len; i+=bufSz) {
   for(j=0;j<bufSz;j+=MTU) {
    ProcessData(buf+off,MTU);
    off += MTU;
    Deallocate_Buffer(buf+off);
   }
}
```

The handler is invoked by an active message from a host or another active switch. The payload of the message contains the arguments, which are written into a predefined address (`ADDRESS2` in this case) and accessed by simply reading that address. The host maps the file to be processed into memory at the location specified by `ADDRESS` by explicitly issuing I/O operations into that memory area and `bufSz` is the size of the disk read request. Since in non-active cases the host allocates a buffer of size `bufSz` before it issues its disk request, the switch can always send a reply to the host with a length of `bufSz` without worrying about storage constraints on the host, simplifying flow control. The `MTU` is the size of the maximum transfer unit. The handler divides a `file_len` amount of data into smaller blocks of size `MTU`, processes each block, and deallocates data buffers occupied by that block when done. While data buffer allocation and mapping is done automatically by the switch hardware, programmers of the active switch need to inform the switch when to de-allocate data buffers. This is done easily by calling the macro `Deallocate_Buffer` with an argument of the ending block address. The hardware will take care of releasing data buffers holding valid mapped addresses less than that end address. All our benchmarks for evaluating active switches use this basic structure. Only the `ProcessData` function is different for different handlers. In our experience, programming active switches is straightforward, made possible by the programming model and the simple nature of most handlers.

3 Active Switch Architecture

This section details the architecture of an active switch. We will show that with the addition of a small amount of hardware, a normal switch can become an active switch, and as we will see in Section 5, the resulting active switch improves the performance of data streaming applications without affecting non-active data streams.

Figure 2 shows the architecture of an active switch with 8 ports. Solid and dashed lines signify data and control lines, respectively. The shaded part of the figure is

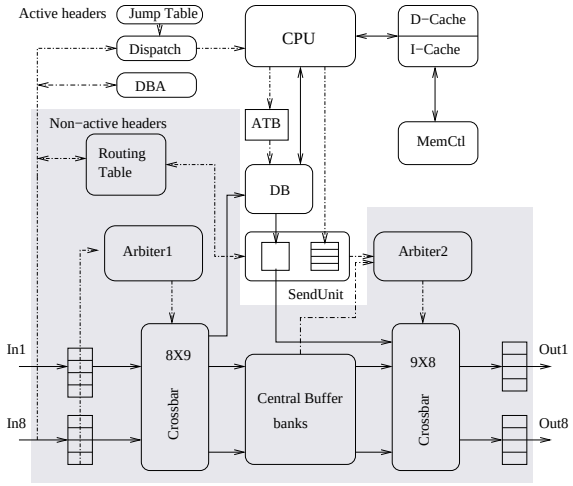

Figure 2. Eight-ported active I/O switch microarchitecture

a normal switch based on a central output queue scheme similar to that in the IBM Switch-3 [23]. An on-chip routing table stores routing information, either on which output port of this switch messages arriving on some input port should leave, or on which output port of the *next* switch this message should leave, depending on the routing scheme used. Unshaded components comprise the "active" hardware added to the conventional switch, including a hardware Dispatch unit that dispatches the active messages to a switch processor; a jump table that stores the starting program counters (PCs) of switch handlers; an ATB that maps physical memory addresses to buffer IDs; data buffers that play an integral role in the processing of incoming and composing of outgoing messages; a data buffer administrator that aids in buffer allocation and de-allocation; and, of course one or more embedded switch processors with separate data and instruction caches. The data buffers are the data interface between the active and non-active portions of the switch and are connected to the Crossbar. The Send unit is used to inform the Crossbar that a data buffer is ready to be sent to one of the output ports.

Any message with a destination of the switch itself is an active message, and the Crossbar guides the message into an unused data buffer in the same way a non-active message is routed to its destination port. The header of the message is extracted and passed to the Dispatch unit in parallel with the copy of the message payload into a data buffer. The Dispatch unit extracts the PC according to the handler ID in the header and schedules the handler on a free switch processor. The Dispatch unit also maps the buffer ID holding the message into a corresponding entry in the ATB according to the destination address field in the header. Because the data and control paths are separate, the switch processor can start processing without waiting for the data buffer copy to complete. In most cases, the switch CPU needs to allocate a data

buffer to compose a new outgoing message. It sends the header of this message to the Send unit, which informs the Crossbar to schedule the message to its destination. Note that the only modification to the non-active part of the switch is logically expanding the Crossbar from $N \times N$ to $(N+1) \times N$. Some switches already have this extra port for management packets.

Let us now explain each part of the active switch in detail. We start with the data buffers—the central staging area for the switch processors. Each data buffer is an independently managed chunk of memory equipped with cache-line based valid bits to allow more parallelism and pipelined data transfers. When a line of data is ready, its corresponding valid bit is set. Accessing an invalid line in a data buffer will stall the switch CPU until that line becomes valid. Incoming data is placed in a data buffer rather than the data cache to keep the cache design simple and its size small, and because data buffers can completely eliminate cold misses and also allow the overlap of data copying and message processing. Buffer management in our programming model is relatively simple. When a message is sent or the switch CPU is done with its processing, the data buffer is freed. Further, because of the streaming nature of active switch applications, only a limited number of data buffers are needed. Most of the applications we consider in Section 5 have only one input data stream and one output data stream and need just 2 buffers. For collective reduction, which needs to process multiple data streams, at least one buffer is needed for every input stream. In our design, we have 16 data buffers, each 512 bytes long (MTU of the network).

Though the data buffers afford many advantages, accessing them with a buffer ID and offset is inconvenient. Further, it makes it difficult for a handler to deal with an object larger than a data buffer. Therefore, we introduce a direct-mapped ATB that maps a memory address into a buffer ID and offset pair, creating the illusion of a flat memory for switch programmers. This again takes advantage of the streaming nature of handlers, since data typically comes into the switch in "order". In our model, each switch CPU has its own 16-entry ATB (one entry per data buffer) that also assists with data buffer deallocation. When a handler needs to release data buffers, it simply provides an address to the ATB, which translates it into the buffer IDs that map all valid addresses less than the given address, and informs the DBA to deallocate the corresponding data buffers. Thus, a programmer need not remember the boundaries of the data buffers and can de-allocate buffer space logically according to the data objects it processes.

Our embedded switch processor model is a single-issue MIPS-like core with extensions to support checking the status of hardware components inside the switch, sending data buffers to other nodes, and requesting or releasing data buffers. To increase the performance of the switch processor, we also include separate instruction and data caches (though as we will see in Section 4 the data cache is quite small). The switch CPU has its own read/write ports to the data buffers so that it can access them in parallel with the switch network ports.

In summary, with the addition of a limited amount of hardware, a normal switch can become an active one without interfering with non-active switch functions, and still keep an easily-understood programming model.

4 Simulation Methodology

In this section we discuss the simulation environment and architectural parameters we use to evaluate our active switch system. The applications themselves are discussed with the results in Section 5. Our simulator models a MIPS-based host processor, its cache subsystem, memory controller, and memory system in detail using a simulator derived from [14]. We model a SAN switch connected to the memory controller with the embedded processor described earlier. The I/O system model is also quite detailed as described below.

In our host processor model a load miss stalls the processor until the first double-word of data is returned, while prefetch and store misses will not stall the processor unless there are already references outstanding to four different cache lines. The processor model also contains fully-associative 64-entry instruction and data TLBs and we accurately model the latency and cache effects of TLB misses. Though the host processor model is relatively simple, what really matters in this research is the relative performance of the host processor and the embedded switch processor, and much of the simulation detail in this study is in the memory, I/O system, and network. We assume that the host processor runs at four times the speed of the switch processor. Specifically, our main 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. For our HashJoin database application, we scale down the cache size so that we can simulate the effect of running problems with large table sizes by running smaller table sizes that we can simulate within a reasonable amount of time. For this class of applications we use an 8 KB primary data cache and a 64 KB secondary cache keeping the same line sizes and associativities.

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 [11] . Our simulated switch models the exact architecture shown in Figure 2. It supports 1 GB/s bidirectional bandwidth and has a routing latency of 100ns,

similar to current InfiniBand switches. The routing algorithm is virtual cut-through and the MTU is configurable (512 bytes for all our experiments). The embedded active switch processor runs at 500 MHz with a 4 KB, two-way set-associative instruction cache with 64-byte lines and a 1 KB two-way set-associative data cache with 32-byte lines. Both caches are small and simple—supporting only one outstanding request.

Our I/O subsystem includes a TCA, an ultra-320 SCSI bus, and simple disks. The SCSI bus models the overhead of arbitration and selection transactions and has a peak throughput of 320 MB/s. The disk model includes three timing related parameters: seek time, rotation speed and peak bandwidth. For all the experiments in this paper, we use two disks with a total peak bandwidth of 100 MB/s and we assume a sequential access pattern because most of our applications deal with large files. The network interface models an InfiniBand HCA connected directly to the memory controller and implements a queue pair interface with the user program. Each network link uses credit-based flow control and we use the Raw packet format as described in the InfiniBand specification. The header is 128 bits, including a 64-bit active header that contains a 6-bit message handler ID field and a 32-bit address field to which the data buffer storing the packet on the active switch is mapped.

I/O-related operating system overhead is the only place where our simulator does not model events in detail and instead charges fixed latencies derived from empirical results from real systems. Though our simulator runs all the library code involved in I/O accesses, it does not execute the lowest-level operating system code. We account for I/O-related operating system overhead by charging 30us of fixed cost per request and 0.27us/KB for each unbuffered disk request. These numbers were obtained from measurement and calculation and were validated against measurements presented in [9].

5 Applications and Results

In this section we present nine applications that can benefit from active switches in system area networks. Different applications may benefit from different aspects of active switches. We will explain in detail the important features for each application in our analysis below. We differentiate four cases for each benchmark. The case running only on the host processor, using non-active SAN switches is called "normal" or "normal+pref" if two outstanding I/O requests are issued. Similarly, the case using both the host processor and active switches is denoted "active" or "active+pref" for the case where there are two outstanding I/O requests. There are two figures for each application. In the first, the performance of these four cases is shown for three metrics: overall execution time, normalized to the "normal" case; host processor utilization, $(1 - idle\ time)/execution\ time$; and host I/O traffic, the total amount of data transferred in/out of the host, also normalized to the "normal" case. In the second figure, the execution time is broken down into CPU busy, cache stall, and idle time. For normal cases, only the host execution time breakdown is shown, denoted as "n-HP" and "n+p-HP" ("n" for normal; "p" for prefetch; "HP" for host processor). For the two active cases, both the host and switch CPU execution time is broken down. We use "a+SP" for the switch CPU in the "active" case and "a+p-SP" for the "active+pref" case.

In Table 1 we summarize the applications and the problem sizes that we simulate. We present descriptions of each application with an analysis of the results below.

Table 1. Applications and Problem Sizes

Applications	Input Data Size (Bytes)
MPEG filter	2202640
HashJoin	16M×128M
Select	128M
Grep	1146880
Tar	4M
Parallel sort	16M
MD5	256K
Collective Reduction	512

MPEG-filter. MPEG-filter is a video file filtering application. It can filter video streams according to the bandwidth constraints on the customer end. This particular benchmark is from the Distributed Multimedia Research Group of Lancaster University, UK. It can perform many kinds of filtering like removing certain types of frames, removing high frequencies, or re-quantization. We use this benchmark to show that with the appropriate partitioning of a program into host and handler portions, active switches and hosts can cooperate in a pipelined fashion to improve performance. In our experiments, we perform two filtering tasks. One is frame filtering: all B-type and P-type frames are filtered out, leaving only I-type frames. The other is color reduction, which reduces a colorful I-type frame to a mono frame that needs decoding and re-encoding. The first filter performs only header checking and not only involves less computation but also filters input data and thus is very appropriate for executing on the switch. If an input video file has many B-type and P-type frames, the handler can greatly reduce the data sent to the host. Color reduction is placed on the host side since it needs decoding and re-encoding, which are compute-intensive. For the normal cases, all tasks are done on the host. The input video file is 2202640 bytes long consisting of I-type frames and P-type frames. About 63.5% of the total data are P-type frames. All I/O requests are made in blocks of 64 KB.

Figure 3 shows the overall performance for each of our four configurations and Figure 4 gives a breakdown of

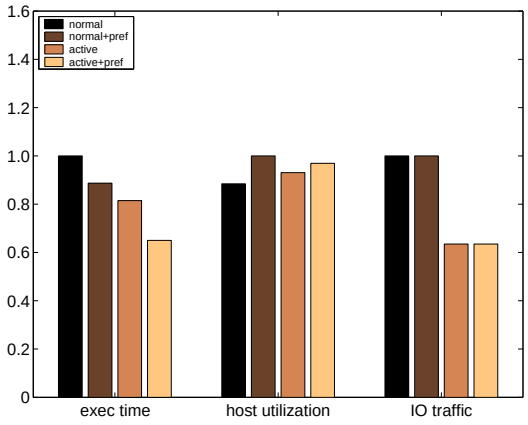

Figure 3. Filter

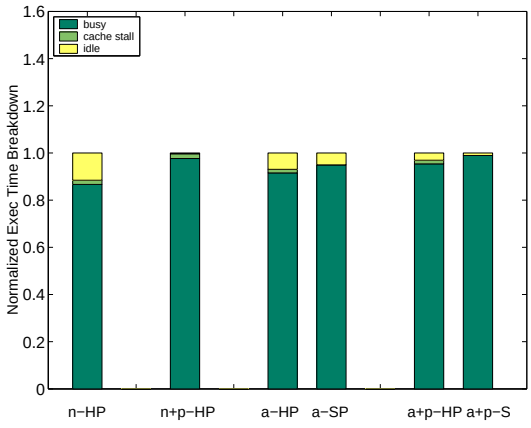

Figure 4. Filter breakdown

execution time. The "normal" case performs the worst partly because of I/O stall time. "normal+pref" is 1.13 times faster than "normal" because it can overlap the computation and disk I/O latency and fully utilize the host CPU. The two active cases achieve a speedup of 1.23 without prefetching and 1.36 with prefetching, respectively, in comparison to the corresponding normal cases. "normal+pref" has the highest host utilization because prefetching makes it completely computation-bound. The two active cases manage to keep both the host and switch CPU busy, whereas the "normal" case has the lowest utilization because of its synchronous disk I/O stall time. For data filtering, both active cases reduced the data sent to the host by 36.5%. Figure 4 explains the origin of the speedup in the active cases. The switch CPU is almost fully utilized, achieving a balanced computing pipeline with the host CPU. In cases where a perfect partition can not be achieved, we would like the switch CPU under-utilized without blocking the host CPU so that active cases will perform better than normal cases. That is the case for all the following applications except MD5, which shows the effects of an unsuccessful application partitioning.

HashJoin with bit-vector filter. Both HashJoin and Select (which follows) show improved performance in active switch systems because of reduced cache stall time on the host CPU due to the filtering of unnecessary records in the switch. When the record size is bigger than the secondary cache size, significant amounts of L2 misses occur for database operations [2]. Active switches can reduce those cold cache misses by filtering out unrelated records inside the switch. Join is a frequently used SQL operation, and a hash-based algorithm is one of the most efficient joins. A further optimization for HashJoin is to use a bit-vector to filter out unmatched records before executing the join algorithm [10]. Bit-vector filtering works in the following way: prior to the initial scan of relation R, which is the smaller one of the two relations to be joined, a bit-vector is initialized by setting all bits to 0. As each R tuple's join attribute is hashed, the hashed value is used to set a bit in the bit-vector. Then as relation S is scanned, the appropriate bit in the bit-vector is checked. If the bit is not set, the tuple from S can be safely discarded. In our experiments, we assume that memory is large enough to hold the smaller relation R. We scaled down the input sizes of R and S to 16 MB and 128 MB, respectively, and the size of the host CPU caches by a factor of eight. Thus we can effectively simulate the behavior of a join of two tables with sizes of 128 MB and 1 GB, respectively. Since the data cache of the switch is already only 1 KB and the bit-vector we use in simulation is around 128 KB, we did not scale down the switch data cache. The reduction factor of bit-vector filtering is 0.24 and the record size is 128 bytes. In the active case, the bit-vector is stored in the switch while the relation R passes through the switch. When relation S is scanned, the active switch filters unmatched records from S according to the bit-vector and sends out those records to the host with the corresponding bit set in the vector. The actual join operation is done by the host.

Figure 5 shows the performance of HashJoin for our four configurations and Figure 6 gives a detailed breakdown of execution time. As in MPEG-Filter, the "normal" case performs the worst because of I/O stalls. Prefetching helps "normal+pref" to overlap the computation and I/O latency. Without prefetch, active switch case has a speedup of 1.10 over the normal case while the peformance is the same for the two prefetch cases. Still, active cases have the smallest host utilization because they offload part of the work to the switch, leaving the host CPU free for other tasks. The "normal+pref" case has the highest utilization because it overlaps computation with its I/O latency. Both active cases also reduced the amount of data sent to the host by 76%. Figure 6 shows that the active switch reduces the number of cache misses. Because the hash table is much larger than the L2 cache and the hashing operation lacks locality, the

cache stall time comprises a significant part of the total execution time—27.6% for the "normal+pref" case. In the active cases, the switch uses its on-chip data buffer to filter out those unmatched records and the host cache miss stall percentage of "active+pref" is only 16.1%. The switch CPU also suffers from cache misses because the bit-vector is too big for its limited L1 data cache and it does not have an L2 cache. However, this impact is small.

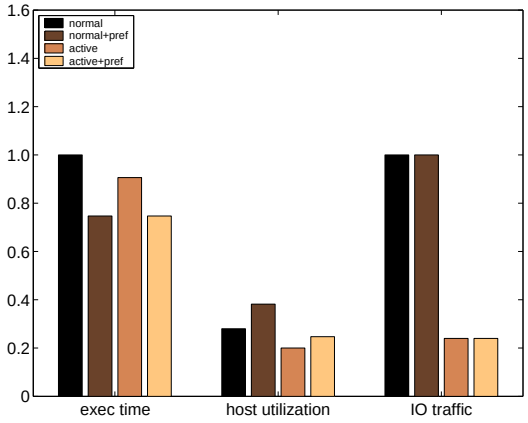

Figure 5. Join

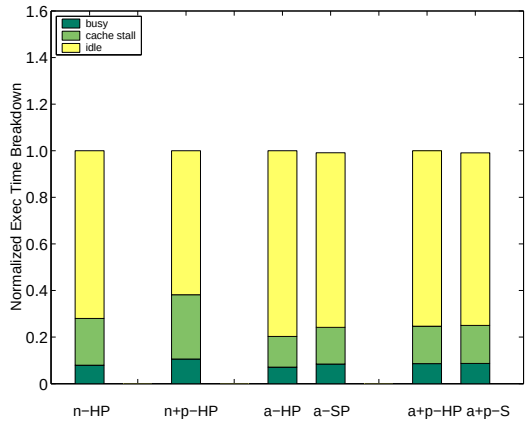

Figure 6. Join breakdown

Select. Our database Select is a sequential range selection that checks if one integer field of a record falls within a specific range. The input data table has a size of 128M bytes with the same configuration as in HashJoin above. In the active cases, selection is done inside the switch and the host CPU just counts the number of matching records.

Figure 8 shows the breakdown of execution time. Again, the "normal" case performs the worst because of high I/O stall time while the other three have almost identical performance because they can overlap the computation with I/O latency and the execution time is determined by the I/O subsystem. All host utilizations are small, indicating that this application is I/O-bound. However, the average host utilization in the two normal cases is 21 times larger than that in the two active cases, which mainly comes from a reduction in host cache misses. The I/O traffic of active cases is 25% of non-active cases. Though the "Normal+pref" case performs equally well in terms of execution time, Figure 8 clearly shows the reduction in cache misses for the host CPUs in the active cases. Since L2 cache miss penalties in modern processors are large and growing, active switches can be beneficial here. Again, there are almost no cache misses for switch CPUs since most of the data processing happens in the data buffers.

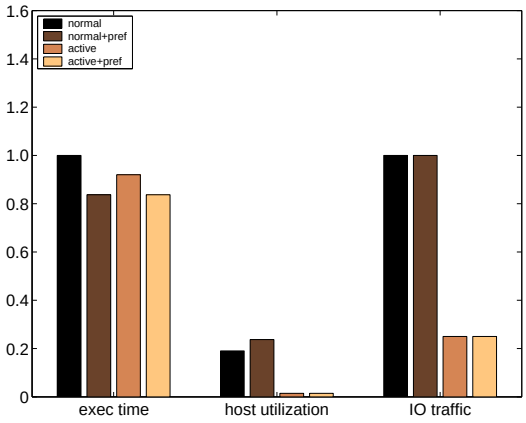

Figure 7. Select

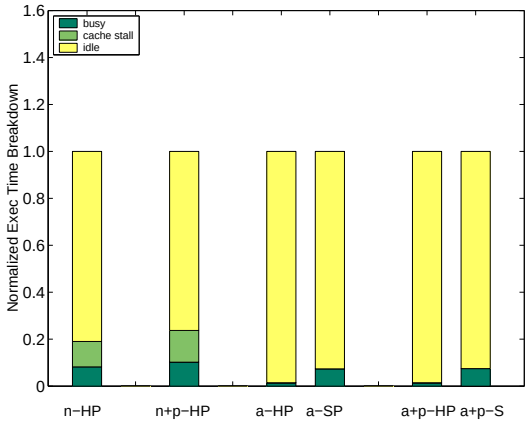

Figure 8. Select breakdown

Grep. Grep and the following application, Tar, show that executing a portion of the application on the active switch can reduce both host utilization and total execution time. We use GNU Grep 2.0, a popular utility that searches input files for lines containing a match to a specified pattern. A basic Grep execution follows three steps: parsing command-line options; setting up a DFA (Deterministic Frontier Analysis) structure; and searching. Our active version of Grep leaves the first step on the host and executes the other two steps on the active

switch. In our experiment, Grep searches only one file for the simple string "Big Red Bear". There are only 16 matched lines out of a file with a size of 1146880 bytes. The I/O request size is 32 KB.

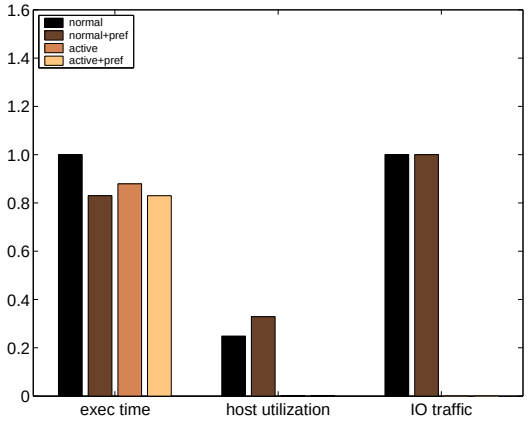

Figure 9. Grep

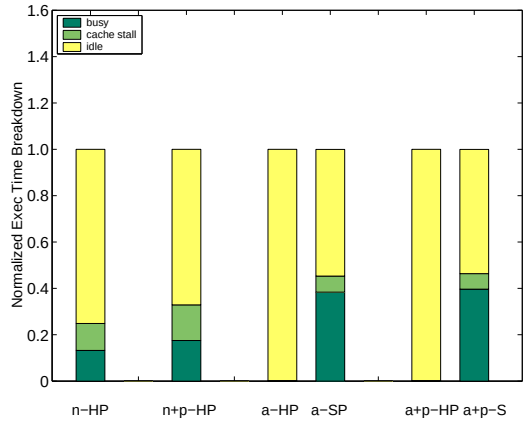

Figure 10. Grep breakdown

Figure 9 shows the overall execution time, host utilization, and host I/O traffic of Grep and Figure 10 gives a breakdown of total execution time. Without prefetching, the active version runs 1.14 times faster than the normal version because the Grep handler can start searching as soon as the first data enters the switch, while the "normal" version has to wait for the entire 32 KB chunk. Prefetching helps "normal+pref" hide the I/O latency and performs better than the "active" case because the latter can not fully hide the I/O latency. The "active+pref" case still performs best because of its proximity to the I/O devices and corresponding lower overhead to initiate I/O requests. As there is very little to do for the host in the active cases, they have utilization close to 0. As for I/O traffic, the handler only transfers back to the host the 16 matched lines, filtering almost all the data. Figure 10 confirms that with the parameters we simulate, Grep is an I/O-bound application.

Tar. In our experiments, we use Tar with "-cf" options that create an archive file from a set of input files. We partition GNU tar as follows: the host portion of active Tar is responsible for parsing the command-line options and generating a header for each input file. The headers are stored into the output tar file. The handler on the active switch reads in the input files and outputs them directly to the archive with the headers created by the host. The handler does not need to perform any processing on the input data packets. It redirects the output tar file to a remote node, completely bypassing the host. Figure 11 shows the performance of Tar under our four configurations and Figure 12 shows a breakdown of total execution time. Similarly, the "normal" case performs worst because of I/O stall time. The other three cases have almost the same execution time, but the active cases have close to 0 host utilization, allowing the host to perform other tasks. Since the host is bypassed, the I/O traffic from the host consists only of the headers for each input file, which are only 512 bytes long. In Figure 12, we can see that the low host CPU utilization in the active cases does not come from offloading the workload to the switch. Rather, most of the busy time in the normal cases is disk I/O-related overhead like interrupt processing, all of which is eliminated in the active switch version.

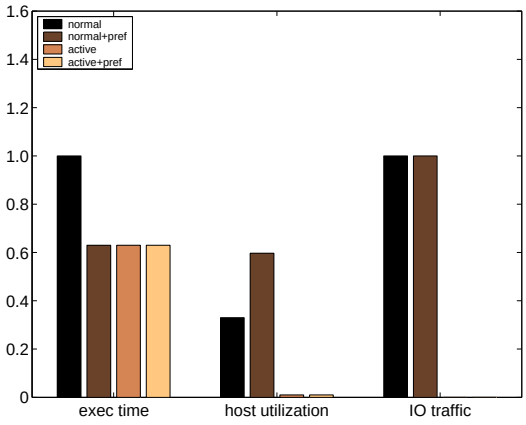

Figure 11. Tar

Parallel Sort. We use Parallel Sort to show how an active switch can redistribute the data according to some criteria and reduce both host CPU utilization and network traffic. Our Parallel Sort works in the same way as a one-pass parallel sort on data with a unified key distribution [3]. Each participating host reads in a portion of the data and performs data redistribution according to a range assigned to each node. After this stage, each node sorts its local data using any sorting algorithm. In this experiment, we sort 16M data items on 4 nodes. The data format follows the Datamation benchmark where each record is 100 bytes long with a key of 10 bytes. For

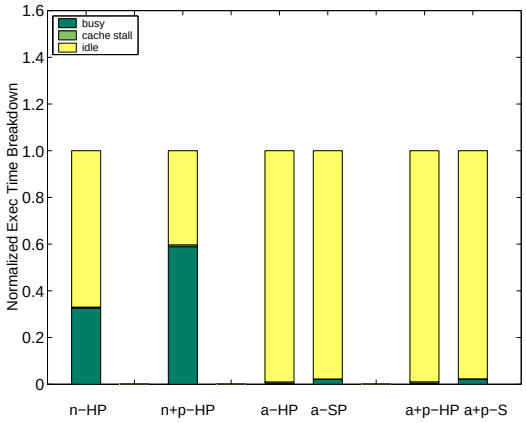

Figure 12. Tar breakdown

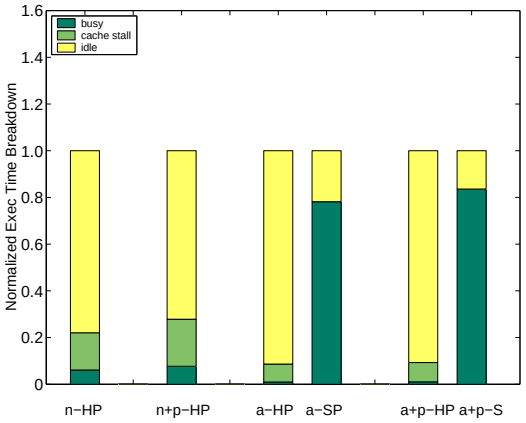

Figure 14. Sort breakdown

the normal cases, each node reads in 1/4 of the total data and then redistributes the data among all the nodes. For the active cases, the redistribution is done by the switch handler so that each node only gets the records assigned to it. Our experiment only simulates the data distribution phase since there is no difference between the active and normal cases in the sorting phase.

Figure 13 shows the performance of Parallel Sort and Figure 14 breakdowns the execution time. The results are similar to those of Grep and we will not repeat the analysis details here. It is worth pointing out that by distributing the input data via the active switch, the amount of data in/out of each node in the active cases is only 40% of that in the normal cases for 4 nodes. The limit is 1/3 following the formula $p/(3p-2)$ where p is the number of participating hosts.

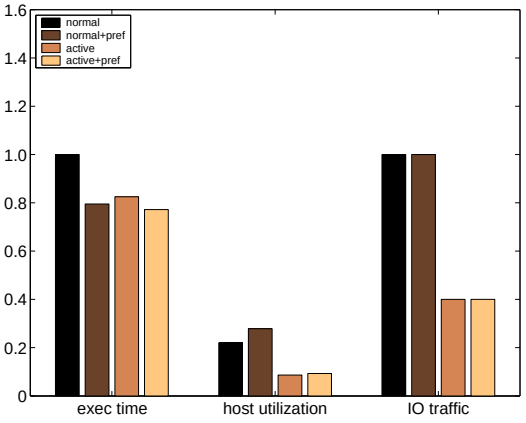

Figure 13. Sort

MD5. MD5 calculates the message digest of a set of input files. For MD5 the active switch cases are slower than than the normal cases. Total execution time is longer when using active switches because it is difficult to find an appropriate partitioning of this compute-intensive code between the host processor and the active switch that is simple enough for the switch to run, yet not one where the switch performs *all* the computation. However, we will re-visit the MD5 application when we discuss multiple embedded switch processors per active switch.

Collective Reduction. In addition to improving I/O-intensive applications, active SAN switches can also improve the performance of the SAN when used as a parallel cluster, by intelligently processing messages between compute nodes. We introduce a Collective Reduction application that shows the power of computation in active switches for some communication patterns. Collective reduction is one kind of collective communication that is at the heart of many parallel algorithms and applications. There are three main types: Distributed Reduce, Reduce-to-all and Reduce-to-one. All three reductions combine the vector on each compute node using a specified operation (often maximum, minimum, sum, product, or logical bit-wise operations). The difference among them lies in how the result vector is redistributed. For Distributed Reduce, each node gets part of this result vector; for Reduce-to-one, there is no need for redistribution; for Reduce-to-all, each node gets the whole result vector. Table 2 shows the two reductions that we use.

Table 2. Collective Reduction

Operations	Before	After
Distr. Red.	$x_0^{(0)} \ \dots \ x_0^{(p-1)}$ $\vdots \qquad \vdots$ $x_{p-1}^{(0)} \ \dots \ x_{p-1}^{(p-1)}$	y_0 $\ddots$ y_{p-1}
Reduce-to-one	$x^{(0)} \ \dots \ x^{(p-1)}$	y

For small vectors, if the communication cost of a message of length n is modeled as $\alpha+n\beta$, the lower bound for the latency of collective reduction on p compute nodes is $\lceil log_2(p) \rceil (\alpha + \lambda)$ where λ is the computation cost for reducing two vectors [22]. With active switches, we propose a new way to do collective reduction that can beat this lower bound. It works in a straightforward way: if all the compute nodes are connected with one switch (a small system), each node sends an active message containing

its own vector to the switch. The switch does the reduction and sends the result vector to the appropriate place according to the type of the reduction. The resulting latency is $\alpha+\gamma$ where γ is the cost added by the switch. The γ in the case of combining 8 512-byte long vectors in our model is close to the fixed overhead of message communication α. The active switch can achieve this small cost because it can access messages coming from all the ports in parallel and has little overhead to start the computation and access message data. Further, the switch can start computation without waiting for the whole message to be copied into the data buffer.

In the case of a system with multiple active switches, the active switch systems also have better scalability than normal systems. We can organize the switches logically in a tree and have each leaf switch combine the vectors from compute nodes connected to it and send the result vector to its parent switch. Finally, the root switch receives the final result vector and sends it to the appropriate nodes according to the type of reduction. Because of the active switch can overlap the switch CPU execution with its duties as a normal switch, the whole latency will be $\alpha + \gamma + \lceil log_{N/2}(p) \rceil \delta$, where N is the number of ports on the switch and assuming that half of the ports of leaf switches are used to connect to compute nodes. δ is the incremental cost added by each level of switch. Active switch systems scale better than normal systems because its scaling factor is $log_{N/2}(p)$ instead of $log_2(p)$.

We only show results for Reduce-to-one and Distributed Reduce here since the results for Reduce-to-all are similar to those for Reduce-to-one. Figure 15 and Figure 16 show the execution time for both the normal and active cases for these two benchmarks. The normal case implements a minimum spanning tree algorithm and the operation used by the reduction is addition. For Reduce-to-one, the switch that calculates the final vector sends it back to node 0. For Distributed Reduce, we use another handler that does nothing but re-distribute the final vector among the nodes. We assume each switch has 16 ports and 8 of them are used to connect compute nodes. Each node has a vector of size 512 bytes. The message receiver uses polling instead of interrupts, which favors the normal case since active switches can eliminate most of the interrupts. Results are shown up to 128 nodes. We see that active switch systems achieve speedup up to 5.61 for Reduce-to-one and 5.92 for Distributed Reduce.

Multiple Switch Processors. All simulation results above assume there is only one switch processor per active switch. It is natural to add more processors inside a switch to take advantage of coarse parallelism from multiple data streams. Next, we take the earlier MD5 application and show the performance impact of adding more embedded switch processors. In our current design, we support up to 4 switch processors per active switch. Each

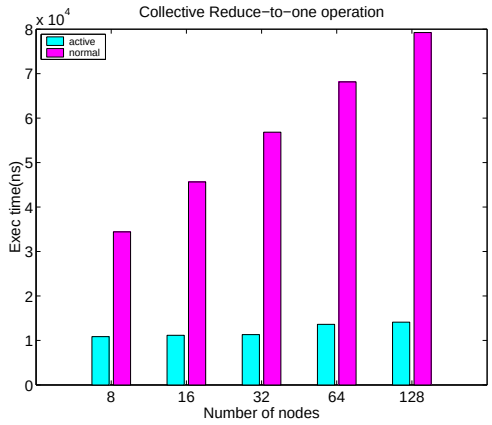

Figure 15. Collective Reduce-to-one operation

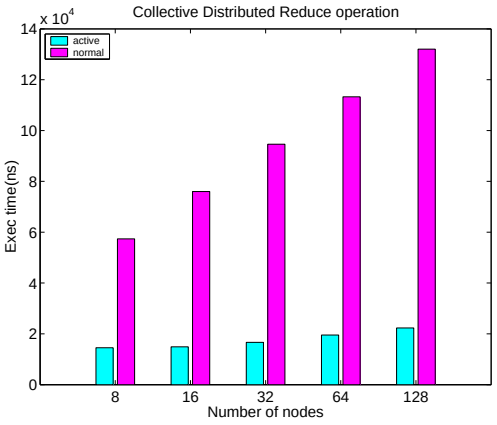

Figure 16. Collective Distributed Reduce operation

switch CPU has its own data and instruction caches.

The Original MD5 uses block chaining to ensure sensitivity to block order that prevents arbitrary parallelism. MD5 can be slightly altered 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. To support multiple switch CPUs, we add a switch CPU Id field in the message header to aid the Dispatch unit. Figure 17 shows the results for this modified MD5 algorithm using 2 and 4 switch processors per active switch. For the purpose of comparison, the results with one switch CPU and the normal case are also included. Instead of an overall slowdown, the active switch system now has a speedup of 1.18 with 4 CPUs and I/O prefetching and 1.50 with 4 CPUs and no I/O prefetching.

6 Related Work

The use of active or intelligent devices to offload tasks from the host processor has been widely explored by researchers. This work can be roughly divided into two classes. In the first class, device functionality is limited

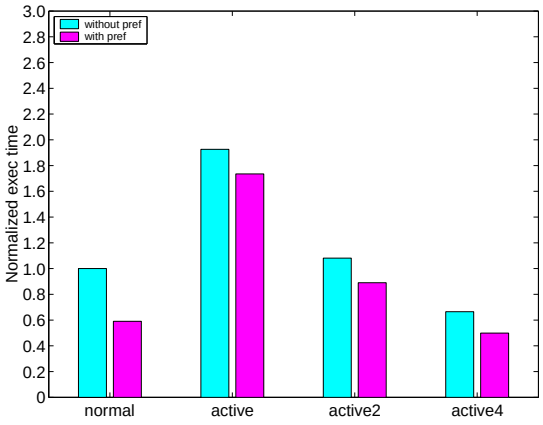

Figure 17. Performance of MD5 with multiple switch CPUs

in the sense that they either have very specific functions or execute only system codes and are not user-programmable. The I/O processor in the IBM 360 allowed users to download *channel programs* that made I/O requests on behalf of the host programs [20]. Network attached storage devices use embedded processors to implement some system-oriented functions like quality-of-service [6] or perform some file system and security functions [13]. There are also many network devices in this class. For example, the Alacritech NIC can offload part of TCP/IP stack processing from the host processor [4].

The second class of devices is less limited and fully user-programmable. Our active switch design falls into this class. There are also many active disks proposals that belong to this group. Among them, Riedel et al. [21] propose a disk model with an embedded processor and evaluate its performance on data mining and image processing applications; Anurag et al. [1] propose a similar active disk system that includes a disk OS and a streaming programming model; Keeton et al. also propose a similar model [17]. All these efforts center on taking advantage of the large number of disks in servers to form a powerful parallel computing engine. There are network devices in this class as well. The Myrinet NIC has embedded processors that can execute user programs. Many research efforts take advantage of this computing power in different situations [5, 7, 12].

The main differences in our approach are that the intelligence lies in the switch rather than in the end devices, and the switch architecture contains customized hardware to separate data from control and improve switch throughput. The location of our active switches within the system yields several advantages over active I/O devices. For one, while active I/O devices can necessarily only improve performance for their particular type of traffic (disk I/O for active disks, network traffic for active NICs), the position of the active switch allows it to potentially improve not only all types of I/O traffic, but as we have shown, even improve host-to-host intra-cluster network traffic. In addition, since scaling to larger systems requires adding more switches, active switch-based systems naturally scale with the machine size. Finally, the cost of the embedded switch CPUs in active switches can be amortized across multiple I/O devices. Since processor performance is increasing faster than individual disk or network I/O performance, it will be possible to actively process four streams (for example) from four passive I/O devices with a single switch, rather than investing in four active I/O devices. If active I/O devices do become prevalent, they can also be used within our active switch system, creating a two-level active I/O system.

Researchers have also explored so-called active networks—a novel approach to network architecture in which the switches of the network perform customized computations on the messages flowing through them. Though this idea is similar to ours in some respects, its main focus is in supporting or introducing new services for wide-area networks using existing network infrastructure. The long distance nature and lack of centralized control in these proposals result in difficult implementation problems. Our focus is on much smaller, more tightly-coupled and managed system area networks and how to improve the performance of the servers built from these networks.

7 Conclusions

In this paper, we propose an active switch architecture for system area networks that can intelligently process messages passing through it on one or more user-programmable embedded processors. Executing code on the SAN switch can reduce host utilization; reduce network bandwidth, host I/O interface bandwidth, and host-memory bandwidth; and improve overall performance either by offloading computation from the host processor or by filtering unneeded data. We also show how active switches are versatile enough to improve host-to-host, host-to-device, and device-to-host communication that can not be done by intelligent devices alone.

We propose a detailed hardware design of an active switch that can seamlessly integrate into existing conventional switch architectures without interfering with non-active messages. A key feature of our switch is the use of on-chip data buffers that eliminate much of the need for large private data caches and avoid wasteful memory copies. The cache line-based valid bits and the separation of the control and datapath of the embedded switch CPU allow more parallelism and are crucial to both the performance and scalability of our system.

To program an active switch, we adopt a hardware/software co-design approach that hides much of the details of the switch hardware from the programmer, creating a familiar programming environment. The partition of a program into host and handler portions is the most

important step. A good partition is one where the handler is small, can filter data, and where both the host and handler code can work in a balanced, pipelined fashion.

To evaluate our active switch concept and hardware design, we developed an execution driven simulator that models all the major hardware components in detail. We show execution time, host utilization, and bandwidth results for 9 benchmarks that demonstrate numerous benefits of active switches—up to 5.9 speedup for some applications. Even where there is little or no speedup, reductions in host utilization and system bandwidth requirements allow for other tasks to be performed concurrently. Thus, active switches can play a key role in improving overall throughput in modern multi-programmed servers.

Acknowledgments

This research was supported by Cornell's Intelligent Information Systems Institute and NSF CAREER Award CCR-9984314.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, October 1998.

[2] A. Ailamaki et al. DBMSs on a modern processor: Where does time go?, In*International Conference on Very Large Databases*, September 1999.

[3] A. C. Arpaci-Dusseau et al. High-Performance Sorting on Networks of Workstations, In *SIGMOD*, pages 243-254, May 1997.

[4] Accelerating Server and Application Performance. http://www.alacritech.com/html/techwhitepaper.html

[5] A. Bilas, C. Liao, and J. P. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *ISCA*, pages 282-293, May 1999.

[6] E. Borowsky et al. Using attribute-managed storage to achieve QoS. In *Presented at 5th Intl. Workshop on Quality of Service*, June 1997.

[7] D. Buntinas, D. K. Panda, and P. Sadayappan. Performance Benefits of NIC-Based Barrier on Myrinet/GM, In*Workshop on Communication Architecture for Clusters*, April 2001.

[8] E. V. Carrera et al. Impact of Next-Generation I/O Architectures on the Design and Performance of Network Servers. In *Proceedings of the First Workshop on Novel Uses of System Area Networks*, pages 27–34, February 2002.

[9] L. Chung et al. Windows 2000 Disk I/O Performance. Microsoft Research Technical Report MS-TR-2000-55, June 2000.

[10] D. J. DeWitt, R. Gerber. Multiprocessor Hash-Based Join Algorithms. In *Proceedings of the 11th Conference on Very Large Databases*, pages 151–164, Stockholm, Sweden, August 1985.

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

[12] M. E. Fiuczynski and B. N. Bershad. SPINE - A Safe Programmable and Integrated Network Environment. In *16th Symposium on Operating System Principles*, October 1997.

[13] G. Gibson et al. File server scaling with network-attached secure disks. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, June 1997.

[14] 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.

[15] M. Heinrich and R. Manohar. Active Fabric: An Architecture for Programmable, Scalable I/O Subsystems. Cornell Computer Systems Lab Technical Report CSL-TR-1998-990, October 1998.

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

[17] K. Keeton, D.A. Patterson, and J.M. Hellerstein. A case for intelligent disks (idisks). In *SIGMOD Record*, 27(3), July 1998.

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

[19] 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.

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

[21] 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.

[22] M. Shroff and R. A. van de Geijn. CollMark: MPI Collective Communication Benchmark, In *Proceedings of the International Conference on Supercomputing 2000*, December 1999.

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

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

[25] 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.