RC23894 (W0603-009) March 1, 2006
Computer Science

IBM Research Report

Evaluating Batching for TCP Offload

Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz,
Erich Nahum, John Tracey
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Evaluating Batching for TCP Offload

Doug Freimuth, Elbert Hu, Jason LaVoie,
Ronald Mraz, Erich Nahum, John Tracey
IBM T. J. Watson Research Center
Hawthorne, NY, 10532
{dmfreim,elbert,lavoie,mraz,nahum,traceyj}@us.ibm.com

Abstract

Ever increasing demand for performance and scalability in server networking has generated significant interest in offloading TCP processing to network adapters. The benefits of TCP offload result from its potential to reduce performance-limiting operations such as interrupts, cache misses, and I/O bus crossings. Exploiting this potential, however, is not easy. Design choices that improve performance for a given hardware configuration, workload or set of network characteristics can reduce performance under different conditions. We have evaluated a TCP offload prototype's ability to reduce I/O bus crossings focusing on the impact of batching interactions between the host and adapter. Our analysis reveals that latency and the number of bus crossings are sensitive to some key parameters related to batching. We demonstrate the importance of these parameters to the viability of any TCP off-load design.

1 Introduction

Offloading TCP processing to the adapter is beneficial because of the changes it enables in the host/adapter interface. These beneficial changes include less cache interference, reduction in interrupts handled, and reduction in the traffic between the host and adapter when TCP processing is done on the adapter. These changes allow the server to gain efficiency in serving content and have the potential to greatly improve overall performance and scalability.

One trade off implied by offloading TCP processing from the host is the granularity of the requests communicated across the I/O interface. In traditional systems, the hardware device driver forwards low level send/receive operations to/from the adapter. Alternatively, TCP offload implies the host adapter interface protocol is at a higher level where operations, such as ack processing, are handled on the adapter. As an example, only the message to free the original host send buffer is required from the adapter to the host. This is in contrast to the adapter relaying each specific ack packet for host processing.

Servers handle multiple operations concurrently, and in doing so, provide TCP offload implementations opportunities to batch operations to and from the adapter. This leads us to the classic trade off between batching performance gains and latency. The use of level counts and timers to efficiently concatenate operations through batching without impacting performance is highly workload dependent. Incorrect settings can result in negligible benefits or worse, very high latency. As an example, setting these levels/timers too small/short results in negligible benefit to batching. Setting them too high/long and the response time for a given request soars.

Traditional communication adapters have employed batching in several ways. One popular method is interrupt coalescing. The idea is straightforward. Rather than immediately posting an interrupt on every event, such as the arrival of a packet, the adapter delays notification to the host until a predetermined number of events happen. If waiting for enough events exceed some predetermined time-out, then the current set of events is sent to the host for processing.

Another way to exploit batching in a TCP offload design is through the use of descriptor queues to communicate request/response operations across the interface. These queues reside in memory on the host or adapter and head and tail pointers are used to signal additions to these request/response queues. Rather than informing the remote entity (host or adapter) on every operation, the software can batch operations by delaying the update of the head/tail pointers which in turn delays the transfer of the entries in the queue. This can be done with level/timer control parameters.

In each of the above examples, interactions between the host and adapter has been reduced but a penalty in latency may result. This latency can be measured in terms of a client/server response time. Depending on the workload, a given set of batching parameters would offer a decrease in interactions but at the same time have a substantial increase in response time. In other workloads, a carefully chosen batching design would offer substantial reduction in host/adapter interactions with a minimal increase in response time.

The work described in this paper evaluates the effect of batching on a specific TCP offload implementation.

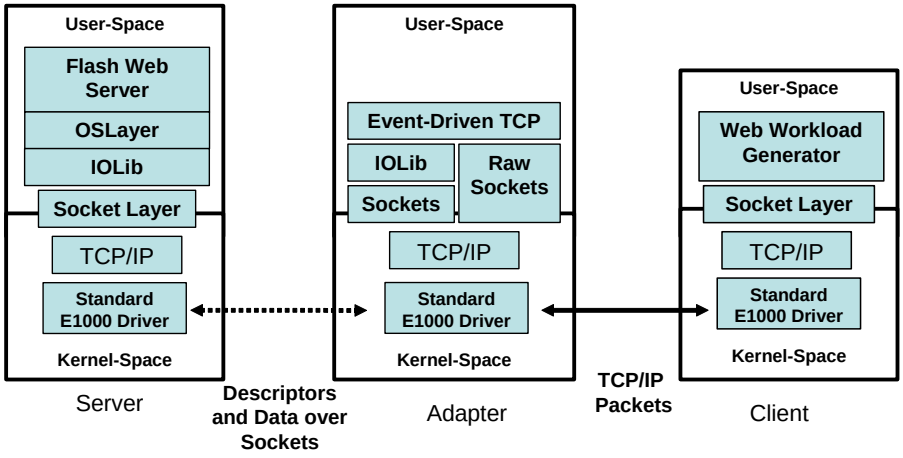

Figure 1: Offload Prototype

We begin by describing the TCP offload implementation used for these experiments and its ability to batch operations between the host and adapter. We then describe some baseline configurations that provide insight into how workload dependent batching can help reduce modeled PCI bus crossings between the host running a web serving application and the adapter supporting a TCP Offload Engine. Specific design points within the prototype are identified and how they limit batching's ability to optimize bus crossings for specific workloads is shown. A detailed experimental analysis provides optimal settings for a specific implementation and workload. Our findings show that batching can substantially decrease bus crossings with minor impact on response time latency within a TCP Offload implementation for specific workloads.

Ultimately, we intend to evaluate the design in a cycle-accurate hardware simulator that will allow us to comprehensively quantify the impact of batching and other design alternatives on cache misses, interrupts and overall performance on current and future generations of hardware.

1.1 Motivation

Earlier work by Freimuth et. al. [7] provided a comparison between a traditional network adapter and a TCP offload prototype. To quantify the benefits of TCP offload, they measured the host to adapter bus crossing traffic in terms of the number of bytes, bus cycles and DMA operations for a set of HTTP requests. Additional comparisons were done for an "optimized" version which provides batching of operations between the host and adapter. These results were encouraging enough for further analysis to quantify the sensitivity of batching for TCP offload. This paper seeks to quantify the effect on response time when varying parameters related to batching within the experimental TCP offload prototype.

1.2 Paper Outline

The rest of this paper proceeds as follows. Section 2 presents a description of the TCP offload design and prototype implementation. Section 3 presents our experimental infrastructure and results. Section 4 surveys and contrasts related work, and Section 5 summarizes our contributions and plans for future work.

2 System Design and Implementation

Here we describe the TCP offload design used in this paper from [7]. The TCP offload prototype offloads the full TCP/IP stack to an intelligent adapter. This is shown in Figure 1. The host and adapter communicate using a higher level of abstraction request and response "descriptor" queues which reside in memory. The control interface allows for the host to command the adapter (e.g., what port numbers to listen on) and for the adapter to instruct the host (e.g., to notify the host of the arrival of a new connection). The control interface can be invoked by conventional socket functions such as: `socket()`, `bind()`, `listen()`, `connect()`, `accept()`, `setsockopt()`, etc.

The data interface provides a way to send and receive on established connections and is invoked by socket functions such as `send()`, `sendfile()`, `write()`, `writev()`, `read()`, `readv()`, etc. Buffers containing data can be much larger than the packet Maximum Transfer Unit (MTU) size. While conceptually they could be of any size, in practice, they are unlikely to be larger than a VM page size.

All DMA data is controlled or "bus mastered" from

the adapter. This is done to eliminate a performance serialization choke point of having the host master all DMA transfers as well as offer consistency in how adapters support bus mastering DMA operations. This includes descriptor queues that reside in memory transferred under the adapter's DMA control.

The interface to the offload adapter is fully asynchronous. The host OS can queue requests to the adapter, continue doing other processing, and then receive a notification (perhaps in the form of an interrupt) that the operation is complete. The host can implement synchronous socket operations by using the asynchronous interface and then blocking the application until the results are returned from the adapter. Asynchronous operation is key for batching to amortize fixed overheads. Asynchrony allows larger-scale batching and enables other optimizations such as polling-based approaches [2, 10].

2.1 Prototype Organization

The TCP prototype is composed of three main components:

- OSLayer, an operating system layer that provides the socket interface to applications and maps it to the descriptor interface used by the adapter;
- Event-driven TCP, the offloaded TCP implementation;
- IOLib, a library that encapsulates interaction between OSLayer and Event-driven TCP.

Currently, OSLayer (OSL) is implemented as a library that is statically linked with the application. Ultimately, it will be decomposed into two independent parts, one of which will be a library, the other of which will be protected in the kernel. This component runs exclusively in the host system. OSlayer is linked with a communication library, IOlib, that abstracts bus interactions (i.e. PCI read and writes) to high level put() and get() semantics.

Event-Driven TCP (EDT) contains nearly all of the offloaded TCP communication functionality. EDT communicates with the host via the IOlib library. Event-driven TCP currently runs as a stand-alone single threaded user-level process that accesses the real network via a raw socket [9].

Event-driven TCP does not notify OSLayer of every packet. For example, instead of informing OSLayer about every acknowledgment, OSLayer is only notified when an entire send operation completes. OSLayer is notified only once for an accept and once for a close. This reduces the number of descriptors (and their corresponding events) to be transferred and processed by the host.

IOLib provides a communication library to the OSlayer and Event- driven TCP code by abstracting the I/O interface to generic put/get calls. This Put/Get abstraction allows for ease of porting the offload prototype to any number of bus, fabric or serial communication interfaces. Thus, only IOLib needs to understand the specific properties of the underlying communication link, while the calls within OSLayer and Event-driven TCP remain unchanged.

The IOLib put/get library components have an asynchronous queuing interface for sending and receiving data. Communications support for the put/get interface can be provided by several types of communication: including shared memory and message passing interfaces.

The current IOLib implementation communicates via TCP sockets, but the design allows for communication over a PCI bus or other interconnects such as Infiniband [3]. Obviously, communicating with a TCP implementation via TCP limits performance and would not be viable for production use. It also provides an excellent vehicle for experimentation and analysis and allows us to measure bus traffic without having to build a detailed simulation of a PCI bus or other interconnect.

2.2 Descriptors

Descriptors are used for both control and data between OSLayer and Event-driven TCP. In the current implementation, there are mirrored response and control descriptor tables, one host to adapter and another for adapter to host. Separate sets of tables are used for each transfer direction.

There are several control descriptors (SOCKET, BIND, etc.) and two data descriptors (SEND, RECV). Most of the control descriptors are of little interest here; however, CLOSE is worth describing. A CLOSE descriptor is transferred from OSLayer to Event-driven TCP when the application initiates a close. After sending out all of the buffers on the write queue, Event-driven TCP will initiate a close. After the final ACK is sent, a response descriptor is created. In the event the other side closes the connection, a CLOSE command descriptor is created in Event-driven TCP and sent to OSLayer. A CLOSE response descriptor is not needed in this case; OSLayer just notifies the application and cleans up appropriately.

When the application calls send, the SEND descriptor is transferred from OSLayer to Event-driven TCP. It contains the address and length of the buffer to be sent. A request for DMA is queued at Event-driven TCP and the next descriptor is processed. After the DMA completes, the event is picked up by Event-driven TCP, and a response descriptor is created with the address of the buffer. This descriptor tells OSLayer the buffer is no longer used. Upon receipt of the SEND response,

OSLayer cleans up the send buffer. Many send descriptors can be sent to Event-driven TCP at once. Buffers described by a SEND descriptor can be up to 4 KB. 4K is chosen since this is the standard page size in most architectures; however, the prototype has the ability to transfer up to 64K bytes.

When receive() is called, the RECV descriptor is transferred from OSLayer to Event-driven TCP. It contains the address and length of the buffer to DMA received data into. If data is available on Event-driven TCP's receive queue, a DMA is queued. Later, after the data is DMA'ed, a RECV response descriptor is created, and the data is available to OSLayer, and Event-driven TCP can free its buffers. If data is not available upon receipt of a RECV descriptor, the buffer is placed on a receive buffer queue for that connection. When data does arrive on the appropriate connection, the buffer is removed from the queue, the DMA takes place, and a RECV response descriptor is created and sent to the host.

All DMA's involving data are initiated by Event-driven TCP. This allows Event-driven TCP to handle the flow up to the host. DMA is not necessarily performed immediately. Since the request for DMA is queued, it may be some time before a response to a descriptor is received by OSLayer.

2.3 Limitations of the Prototype

Certain non-essential descriptors are not yet implemented. An immediate mode descriptor, that is, one with the data in the descriptor, would reduce the number of bus crossings for small transfers. Descriptors for sending status (e.g., the number of available send buffers) and set options could also improve performance and allow more dynamic behavior. Finally, descriptors to cancel a send or a receive are not yet implemented.

A bulk ACCEPT descriptor will allow OSLayer to instruct Event-driven TCP to wait for N connections to be established before responding. A single response descriptor would contain all of the information about each connection. In the appropriate scenarios, this will reduce ACCEPT descriptor traffic.

OSLayer within the TCP Offload prototype is still under development. Many of the socket options not used by Flash web server are not implemented or handled properly. Additionally, having both OSL and EDT execute as user level processes can impact the accuracy of the timers for the experiments. Multiple runs were made to insure low variance of the results.

3 Experimental Results

In this Section, we present the results of the TCP prototype described in Section 2. We first define the variables that are available in the TCP prototype implementation; then describe the measurement capabilities related to batching and the experimental set-up. We then quantify the maximum benefit related to bus crossings that one can expect from batching by using relatively large batching parameters. We next optimize the response time while preserving the maximum benefit in terms of reducing bus crossings. We then vary the number of concurrent connections in addition to the workload.

3.1 Experimental Parameters

Our TCP offload prototype provides several independent parameters related to batching. The two that directly impact batching are the batching level and batching timer. these can be set independently for the host and adapter.

Batching Level and Batch Timer. The batching level specifies the minimum number of descriptors transferred. After the first descriptor in a batch is created, the batch timer is set. If the timer fires before the batching level is reached, the descriptors are transferred. The queue may end up with many more entries than the minimum number by the time the batched data is sent. These can be set independently in OSL and EDT. Typical values of the Batching Timer are 1, 5, and 10 milli-seconds.

Max transfer size. This is the maximum size of a buffer (data or batched descriptors) that can be sent between OSL and EDT. This value must be consistent across EDT, OSL, and IOLib. This variable was statically set to 4K for these experiments and not varied.

Max descriptors. This is the maximum number of descriptors to send per transfer. The default for this is Max transfer size divided by the size of the descriptor (e.g. 4096/32 or 128) because this variable is dependent on the Max transfer size, this variable was fixed for our experiments.

Cork Timer. When corking is invoked, send data is buffered within OSL until the Maximum Transfer Size is reached or a preset timing interval (in configuration file) is reached. Corking can be invoked as a TCP option per connection and the application can "uncork" the data buffers. In standard TCP implementations, the retransmit timer is used for the Cork Timer which can be on the order of a second as described in Stevens [17]. Since the TCP prototype we are using has the retransmit timer offloaded to the adapter, a new "Cork Timer" is implemented in the host's OSL software for this purpose.

When sending a small file, a large cork timer may hurt performance. For a large file, corking should reduce the number of transfers between the host and the card. Typical values for the Cork Timer are 10 milli-seconds or greater but for this investigation corking is set to 1 millisecond meaning no long term buffering of send data occurs in the OSL software.

3.2 Instrumentation

This paper examines the number of DMA's performed and Response Time Latency. Since IOLib is the interface between the host and the adapter, it is a natural place to monitor traffic between the two. To facilitate comparisons to conventional adapter implementations, IOlib is instrumented to measure the number of DMA's across the interface.

Response Time latency of HTTP responses and data transfer time, as viewed from the client, is provided by the httperf web server [11] performance program. Response time is the amount of time between sending the first byte of the request and receiving the first byte of the reply. Transfer time is the amount of time required to transfer the remaining data. When the entire reply fits into a single TCP segment data transfer time is zero, and the total transfer time is the initial response time.

Both EDT and OSL provide a histogram of the number of entries in a queue when each transfer takes place. Examination of the histogram was useful in understanding the limitations of the prototype with respect to optimizing the batching parameters for response time and bus crossings.

3.3 Experimental Setup

We use a simple web server workload to evaluate our prototype. This arrangement is shown in Figure 1. The hardware is a Intel server running a Linux 2.6 kernel for both the host and adapter components colocated in the same box and communicating across the machine's "localhost" communication interface. A Linux workstation serves as our test client connected across a 100 Mb Ethernet Lan to the host and adapter components. The application software consists of the Flash Web server [13] and the httperf [11] client workload generator. Flash, in addition to OSLayer and Event-driven TCP, all run on the same machine for our experiments. We use httperf running on a separate clinet machine to drive the experiments.

We examine three file size transfers: a small (1 KB), a moderate (64 KB), and a large (512 KB) file. This is intended to capture a spectrum of data transfer sizes and vary the ratio of per-connection costs to per-byte costs. We measure the number of transfers in both (send and receive) directions and the number of times a DMA is requested from the bus.

For all experiments, we first create a baseline configuration with Batching Level = 1 and Batching Timer = 1 for both the Host and the Adapter. This provides a starting point to examine selective batching of the host and adapter. This is the minimum amount of batching of at least one entry in the queue making the timer irrelevant. We then vary the batching timer and batching levels on either the host or the adapter and report the optimal configuration for each. These values are then consolidated for batching on both host and adapter. The experiments vary the number of simultaneous connections.

The range of values for Batching Level is 1 to 128 where 1 is the minimum number of entries for any transfer and 128 is the fixed depth of the request/response queues. We limit the range of the Batching Timer from one milli-second to one-half the average response time of a 1K HTTP request. The rationale is that such a long timeout would provide any TCP implementation with ample time to batch as much of the queue entries as possible. A value of zero milli-seconds is used with a batching level of 1 because any entry in the request/response queue is sent immediately.

3.4 Reducing Bus Crossing by Batching

The results presented in Table 1 begin with the baseline for 1K, 64K and 512K single connection entries. The entry for 1K transfer time is zero because the response data at this size is already contained in the initial response time packet. This is independent of batching.

For these experiments the batching parameters were set to the maximum levels to allow the maximum amount of batching. These levels were batching level of 128, batching timer equal to one-half the response time of the complete HTTP request in the baseline experiment. Given all this excess time allowed for batching, this experiment provides us with an indication of the amount of bus crossing reduction one can expect at the expense of response time.

From Table 1, we see a reduction in the normalized bus crossings to 0.77 and 0.94 for 1KB and 64KB experiments respectively. But, from the table we see that the response time is very sensitive to batching timeouts. The effect can be as high as a ratio of 17:1 when using the maximum batching parameters.

3.5 Optimizing Bus Crossing and Response Time

Table 2 provides experimental results where we seek optimal batching to reduce bus crossings with minimal impact to response time for single connections. We are attempting to improve on the numbers provided in Figure 1. In other words, what is the configuration with the best value response time that improves bus crossings over the baseline.

We show that for the 1K case, using adapter batching, we are able to reduce the normalized bus crossings to 0.77 while increasing normalized Response Time to 1.05. This matches our bus crossing best case effort in Table 1 and shows this value can be achieved with only minimal increase in response time.

In contrast, the 64K runs were unable to improve upon the baseline without significant increase in re-

Configuration	1 KB			64 KB			512 KB		
	RT	TT	Xing	RT	TT	Xing	RT	TT	Xing
baseline - minimal batching	1.0	0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Host batching	10.6	0	1.0	5.7	1.0	1.0	5.7	1.0	1.0
Adapter batching	8.17	0	0.77	8.21	1.0	0.94	8.03	1.0	1.0
Host and Adapter batching	17.8	0	0.77	12.93	1.0	0.94	12.07	1.0	1.0

Table 1: Normalized Response Time (RT), Transfer Time (TT) and Bus Crossings (Xing) for Maximum batching parameters across host only, adapter only and host/adapter batching configurations for a single active connection. Using maximum values for batching parameters, this represents the greatest reduction in bus crossings for single connection run.

Configuration	1 KB			64 KB			512 KB		
	RT	TT	Xing	RT	TT	Xing	RT	TT	Xing
baseline - minimal batching	1.0	0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Host batching	1.81	0	1.0	1.93	1.00	1.0	1.09	1.0	1.0
Adapter batching	1.05	0	0.77	1.00	1.11	1.0	1.03	1.0	1.0
Host and Adapter batching	1.82	0	0.77	1.99	1.00	1.0	1.17	1.0	1.0

Table 2: Normalized Response Time (RT), Transfer Time (TT) and Bus Crossings (Xing) for different batching configurations to optimize bus crossing with minimal impact to Response Time for single active connections.

Configuration	1 KB			64 KB			512 KB		
	RT	TT	Xing	RT	TT	Xing	RT	TT	Xing
baseline - minimal batching	1.0	0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Host batching	2.27	0	0.67	2.05	1.29	0.95	1.54	1.04	0.97
Adapter batching	1.05	0	0.77	1.06	1.00	1.00	0.89	1.00	1.00
Host and Adapter batching	2.44	0	0.48	2.11	1.30	0.90	1.64	1.06	0.97

Table 3: Normalized Response Time (RT), Transfer Time (TT) and Bus Crossings (Xing) for different batching configurations to optimize bus crossings while minimizing Response Time for five active connections.

sponse time. Representative runs for adapter and host batching are provided as a reference. Finally, the 512K experiments shows varying the batching parameters has no effect on response time.

3.6 Concurrent Connection Results

Table 3 shows another set of experiments for runs with 5 concurrent connections. Having multiple active requests produces additional traffic which provides more opportunities to batch descriptor queue entries across the IO interface. As before, we use httperf to generate the traffic with the number of connections set to 5 and the Rate set to 5. An average of response time and transfer time over multiple runs are provided. As before, we provide baseline runs using minimal batching configuration parameters for 1K, 64K and 512K file sizes.

Running a large sample of runs while iterating configuration parameter values, we were able to provide representative configurations for optimal Response Time with reduced bus crossings and a configuration for optimal bus crossings (Xing) independent of Response Time reduction for the 1K, 64K and 512K workloads.

For the 1K and 64K workload we found adding batching on the adapter produced the best response time result with reduced bus crossings. Alternatively, adding batching to the host side produced the best value for bus crossings independent of response time. As a control, we provide an additional run with batching on both the host and adapter.

The 512K workload was not as definitive in the advantages of batching on either the host or the adapter. The number of entries required in the request/response queues to transfer multiple 512K data transfers will exceed the current 128 Max Descriptor entry limit. In this case, batching is limited by the Max Descriptors the prototype can send across the IO bus.

3.7 Overall Observations

When sending small amounts of data (1K) for an HTTP response, this has the greatest potential benefit for batching. This is because for larger amounts of data, much of the response data is generated in bulk via sendfile and is already batched in our implementation. As multiple data descriptors are entered in queues, these are batched by default. Additionally, the effect of our batching parameters on larger 512K files is minimal because the number of potential descriptors exceed the allowable entries in the descriptor queue.

When sending smaller amounts of data, we find that adding *the correct* amount of batching in the adapter can help reduce bus crossing with little impact on the overall response time. Conversely, adding batching on the host can reduce the bus crossings even further but with a greater penalty in response time.

We also found that transfer time can be impacted when sending a moderate amount of data (64K). Unfortunately, any improvement in this area over the baseline were offset by additional delays in the Response Time.

4 Related Work

Many improvements in server scalability were described more than fifteen years ago by Clark et al. [5]. The authors demonstrated that the overhead incurred by network protocol processing, per se, is small compared to both per-byte (memory access) costs and operating system overhead, such as buffer and timer management. Nearly all of the enhancements described by Clark et al. have seen widespread adoption.

Other related work on TCP offload has generated both promising and less than compelling results [1, 14, 16, 19]. There is scarce information related to the effect of batching data or queue entries from the host to the adapter. The lack of mention of the subject is puzzling since, as we have shown in the paper, an incorrect selection of batching properties in the offload design can easily cripple performance of the TCP offload system.

Performance analysis of current generation network adapters only reveals the characteristics of networking at a given point in time. In order to understand the performance impacts of various design tradeoffs, all of the components of the system need to be modeled so that performance characteristics that change over time can be revealed. Binkert et al. [4] propose the execution-driven simulator M5 to model network-intensive workloads. M5 is capable of full system simulation including the OS, the memory model, caching effects, DMA activity and multiple networked systems. M5 faithfully models the system so it can boot an unmodified OS kernel and execute applications in the simulated environment. In Section 5 we describe the use of Mambo, an instruction level simulator for the PowerPC®, in order to faithfully model network-intensive workloads.

Researchers have shown that interrupts are costly, and generating an interrupt for each packet arrival can severely throttle a system [10]. In response, adapter vendors have enabled the ability to delay interrupts by a certain amount of time or number of packets in an effort to batch packets per interrupt and amortize the costs [6]. While effective, it can be difficult to determine the proper trigger thresholds for firing interrupts, and large amounts of batching may cause unacceptable latency for an individual connection.

Shivam and Chase [16] showed that offload can enable direct data placement, which can serve to eliminate some communication overheads, rather than of shifting them from the host to the adapter. They also provide a simple model to quantify the benefits of offload based on the ratio of communication to computation and the ratio

of the host CPU processing power to the NIC processing power. Thus, a workload can be characterized based on the parameters of the model and one can determine whether offload will benefit that workload. This paper can be seen as an application of Amdahl's Law to TCP offload. Their analysis suggests that offload best supports low-lambda applications such as storage servers.

The current generation of offload adapters in the market have simply moved the TCP stack from the host to the offload adapter without the necessary design considerations for the host and adapter interface. For some workloads, this creates a bottleneck on the adapter [14]. Handshaking across the host and adapter interface can be costly and reduce performance especially for small messages. Additionally, Ang [1] found that there appears to be no cheap way of moving data between host memory and an intelligent interface.

The TCP prototype used in this paper could be modified to incorporate cumulative completion descriptors. Instead of completing each send or receive request individually with its own SEND/RECV complete descriptor, the prototype will have a send complete descriptor that indicates completion of all requests up to and including that one. This change will require zero changes to the descriptors. To do this one simply changes the semantics of the descriptors such that completion of a send/receive implicitly indicates completion of any sends that preceded it. This approach is employed by OE [19], and it appears similar benefits can be achieved in our stack as well.

5 Summary and Future Work

Batching is an important area of study for optimizing the host to adapter interface of any TCP offload implementation. We have shown that it is possible to derive significant benefits through batching within a TCP offload design. Although there is the traditional trade-off between latency (measured as response time) and improved performance (measured in terms of IO bus crossings) we find that we can, in some cases, achieve optimal performance (23% improvement) with very little impact (5%) on latency.

In general, we were not able to show any performance gains for batching of larger amounts of data such as 64K and 512K HTTP transfers. This is because for larger amounts of data, much of the response data is generated in bulk, by way of a call to sendfile as in [12], and consequently is already batched in our implementation because the parameters are really minimum levels for batching operations. But, optimizing performance for small transfers will help typical Wide area network since [18] describes Web Client and Server traffic having an average "flow" of 1K and 9-12K, respectively.

An application of this evaluation would be to optimize a high bandwidth network interface card that resides on a slower IO interface. Such an example would be a 10 Gbit Ethernet adapter that sits in a sub 10 Gbit bandwidth bus, such as a PCI-X [8] or a slower bus interface. Given, that each DMA crossing incurs bus overhead and host system latency, from a performance standpoint, it makes sense to reduce the number of crossings to better utilize this restricted resource.

The parameters used in this study were limited to the Batching Levels and Batching Timers of the host and adapter IO interfaces for brevity in this publication. Additional studies regarding the cork timer, data and descriptor transfer size and maximum descriptor entries sent across the IO interface can add additional insight into batching and TCP offload design trade-offs in general.

Future work includes additional studies with more connections and different rates of connections. Additionally, comparisons for a suite of applications that include network file transfer such as NFS clients and servers, and other web related application are in the works.

Cycle accurate simulators for performance studies under a suite of applications can provide a more comprehensive view of the benefits of batching for TCP offload in general. These are within Mambo, a simulation environment for PowerPC® systems [15]. Mambo allows us to run the OSLayer (host) and Event-driven TCP (adapter) portions of the prototype on distinct simulated processors. This would allow us to determine the hardware resources needed on the adapter to support a given host workload.

Finally, we intend to extend the prototype and simulation to encompass low-level device interaction. This will entail replacing the socket-based version of IOLib with a version that communicates across a hardware interconnect such as PCI or InfiniBand®. This will allow us to predict throughput and latency on simulated next-generation interconnects.

References

[1] Boon S. Ang. An evaluation of an attempt at offbading TCP/IP protocol processing onto an i960rn-based NIC. Technical Report 2001-8, HP Labs, Palo Alto, CA, Jan 2001.

[2] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.

[3] The Infiniband Trade Association. The Infiniband architecture. `http://www.infinibandta.org/specs`.

[4] Nathan L. Binkert, Erik G. Hallnor, and Steven Reinhardt. Network-oriented full-system simulation with M5.

In *Proceedings Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads CAECW*, Anaheim, CA, Feb 2003.

[5] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6), June 1989.

[6] Peter Druschel, Larry Peterson, and Bruce Davie. Experiences with a high-speed network adaptor: A software perspective. In *ACM SIGCOMM Symposium*, London, England, August 1994.

[7] Douglas Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey. Server scalability and tcp offload. In *USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[8] The PCI Special Interest Group. PCI-X specifi cation. `http://www.pcisig.com/specifications/pci_x`.

[9] The libpcap Project. `http://sourceforge.net/projects/libpcap/`.

[10] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[11] David Mosberger and Tai Jin. httperf – a tool for measuring Web server performance. In *Proceedings 1998 Workshop on Internet Server Performance (WISP)*, Madison, WI, June 1998.

[12] Erich M. Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, 10(2):2–11, Feb 2002.

[13] Vijay Pai, Scott Rixner, and Hyong-Youb Kim. Isolating the performance impacts of network interface cards through microbenchmarks. In *Proceedings ACM Sigmetrics*, New York, NY, June 2004.

[14] Prasenjit Sarkar, Sandeep Uttamchandani, and Kaladhar Voruganti. Storage over IP: When does hardware support help? In *USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003.

[15] H. Shafi , P.J. Bohrer, J. Phelan, C.A. Rusu, and J.L. Peterson. Design and validation of a performance and power simulator for POWERPC systems. *IBM Journal of Research and Development*, 47(5/6):641–651, September/November 2003.

[16] Piyush Shivam and Jeffrey S. Chase. On the elusive benefi ts of protocol offbad. In *ACM SigComm Workshop on Network-IO Convergence (NICELI)*, Germany, August 2003.

[17] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: the protocols*. Addison-Wesley, Reading, Massachusetts, 1994.

[18] K. Thompson, G. Miller, and R. Wilder. Wide-area internet traffi c patterns and characteristics. *IEEE Network*, 15(3), November 1997.

[19] R. Westrelin, N. Fugier, E. Nordmark, K. Kunze, and E. Lemoine. Studying network protocol offbad with emulation: Approach and preliminary results. In *12th Annual IEEE Symposium on High Performance Interconnects*, Stanford, CA, Aug 2004.