Modeling Protocol Offload for Message-oriented Communication *

Patricia Gilfeather and Arthur B. Maccabe
Scalable Systems Lab
Department of Computer Science
University of New Mexico
pfeather@cs.unm.edu, maccabe@cs.unm.edu

Abstract

In this paper, we present a new, conceptual model that captures the benefits of protocol offload in the context of high performance computing systems. In contrast to the LAWS model, the extensible message-oriented offload model (EMO) emphasizes communication in terms of messages rather than flows. In contrast to the LogP model, EMO emphasizes the performance of the network protocol rather than the parallel algorithm. The extensible message-oriented offload model allows protocol developers to consider the tradeoffs and specifics associated with offloading protocol processing including the reduction in message latency along with benefits associated with reduction in overhead and improvements to throughput.

We give an overview of the EMO model and show how our model can be mapped to the LAWS and LogP models. We also show how it can be used to analyze individual messages within TCP flows by contrasting full offload (TCP offload engines) with other approaches, e.g., interrupt coalescing and splintered TCP.

1 Introduction

Commodity network speeds are increasing. Gigabit Ethernet is now commonplace. In the near future, 10 Gigabit and 40 Gigabit Ethernet will replace Gigabit Ethernet as the commodity networking technology. Offloading all or portions of communication protocol processing to an intelligent NIC (Network Interface Card) is frequently used to ensure that benefits of these technologies are available to applications.

Shivam and Chase created the LAWS model to study the benefits and tradeoffs of offloading[11]. However, there are no models that address the specific offloading concerns of high-performance computing. We create a model that explores offloading of protocols for individual messages which allows us to consider offloading performance for message-oriented applications and libraries like MPI.

In this paper, we provide a new model, the extensible message-oriented offload model (EMO), that allows us to evaluate and compare the performance of network protocols in a message-oriented offloaded environment. First, we overview two popular performance models, LAWS and LogP and show how the current models do not meet the specific needs of modeling for high-end applications. Second, we review the characteristics necessary for a performance model for high-performance computing network protocols. Third, we introduce EMO which is a language for capturing performance of various offload strategies for message-oriented protocols. We explain a model of overhead and latency using EMO and map EMO onto LAWS.

Finally, in this paper, we develop a case study using EMO to answer the following question. Can we build a suitable TCP offload model that is inexpensive and decreases latency for small messages? Using EMO we demonstrate the concept of overhead hiding for Splintered TCP.

2 Previous Models of Communication Performance

There are two performance models we considered before creating one that was specific to the needs of high-performance computing.

2.1 LogP

LogP was created as a new model of parallel computation to replace the outdated data-oriented PRAM model. It is based on following four parameters.

- L - upper bound on latency
- o - protocol processing overhead

*Los Alamos Computer Science Institute SC R71700H-29200001

- *g* - minimum interval between message sends or message receives
- *P* - number of processors

LogP is message-oriented. The original model assumes short messages, but several extensions to LogP for large message sizes have been proposed [1].

The LogP model[10] found that high-performance applications are linearly sensitive to changes in the overhead of protocol processing and to changes in the gap between messages. High-performance computing is associated with very computationally-intensive tasks. This means that the majority of the resources in high-performance applications must be reserved for computing tasks and communication overhead must be kept to a minimum.

Gap is the time between sending one message and sending the next message or receiving one message and receiving the next message. Gap can be considered a measure of the effectiveness of pipelining messages through a network protocol stack. In the limit, gap provides a measurement of the limiting factor in protocol performance. To increase performance in high-end applications, one must minimize protocol processing overhead and minimize gap.

However, the focus of LogP is on the execution of the entire parallel algorithm and not on the performance of the particular network protocol. The LogP model gives us the understanding that the overhead and gap of communications must be minimized to increase performance of parallel algorithms. It does not, however, give us any insight into how this may be done. The LogP model, therefore is not helpful in the direct design and implementation of protocols that increase performance of high-performance applications, but it does inform us as to which metrics to minimize.

2.2 LAWS

The LAWS model was created to begin to quantify the debate over offloading the TCP/IP protocol. LAWS attempts to characterize the benefits of transport offload. It is based on the following four ratios.

- *L*ag ratio(α). The ratio of host processing speed to NIC processing speed
- *A*pplication ratio(γ). The ratio of application processing to communication processing–how much CPU the application needs
- *W*ire ratio(σ). The ratio of bandwidth when host is at 100% utilization to raw network bandwidth–how much bandwidth is affected by CPU utilization
- *S*tructural ratio(β). The ratio of overhead for communication with offload to overhead without offload–what processing was eliminated by offload

The LAWS model effectively captures the benefits and constraints of protocol processing offload. Furthermore, because the ratios are independent of a particular protocol, LAWS is extensible. When extending LAWS to an application-level library, the application ratio (γ) must reflect the additional overhead associated with moving data and control from the operating system (OS) to the application library, but this is trivial. However, LAWS is stream-oriented and not message oriented. Specifically, it cannot help us to understand how to minimize gap or latency which are primary needs for our model.

LAWS is a good model of the behavior of offloading transport protocols. We provide a mapping from our high-performance message-oriented model to the LAWS model in section 3.3 so we may benefit from the understanding that the LAWS model brings to the question of how and when to offload.

3 Extensible Message-Oriented Offload Model

Neither the LAWS model nor the LogP model help us to evaluate methods for offloading network protocols in the high-performance computing environment. LAWS is not message-oriented and so it does not allow us to model either gap or latency. LogP is not specifically oriented to network protocol performance. We needed a new model of communication in high-performance computing.

3.1 Requirements for High-performance Model

We wanted to create a simple language for modeling methods of offload in order to understand how they relate to high-end applications. In addition to the ability to model latency, gap and overhead, we had three requirements for our performance model. We wanted the model to extend through all layers of a network protocol stack including message libraries like MPI at the application layer. We wanted to model offload onto a NIC as this was our primary focus. We wanted to model behavior in a message-oriented environment.

3.1.1 Extensible

Extensibility is necessary for our model because network protocols are often layered. Layered above the network protocols are more layers of message-passing API's and languages like MPI and LINDA. We developed our model to extend through the layers of network protocols and message-passing API's.

For example, one of the reasons that TCP has not been considered competitive in high-performance computing is

that the MPI implementations are not efficient. The MPI implementations over TCP are generally not well integrated into the TCP protocol stack. A zero-copy TCP implementation still requires a copy in application space as the MPI headers are stripped, the MPI message is matched and the MPI data is moved to the appropriate application buffer. A zero-copy implementation of MPI will require a way to strip headers and perform the MPI match at the NIC level. Again, application libraries like LINDA are implemented on top of MPI. The same process will continue through all layers of the communication stack. We want our model to be extensible so we can capture this behavior.

3.1.2 Offload

Offloading parts or all of the processing of a protocol in order to decrease overhead has been commonplace for years. In the commodity markets of Internet serving and file serving, TCP offload engines (TOEs) are becoming more common as they attempt to compete with other networks like FibreChannel.

In high-performance computing, Myrinet, VIA, Quadrix and IB all do some or all of their protocol processing on a NIC or on the network itself. Offload is an attractive way to keep overheads on the host low. Our goal in producing this model is two-fold. First, we want to provide a way to explore whether smart offloading of a commodity protocol like IP or TCP could eventually make these protocols competitive in the high-end computing arena. Second, we want a tool for exploring offloading in the development of future high-performance computing protocols.

Offloading is the central focus of this model. Like the LAWS model, the goal of this model is to explore the benefits of offloading transport protocol processing. Unlike the LAWS model, we are doing so in the context of message-oriented high-performance applications.

3.1.3 Message-Oriented

We need a performance model that is message-oriented to that we can specifically model and compare methods of offloading that decrease overhead or latency. The message-oriented nature of the model also provides the structure necessary to model gap in a new way.

The message-oriented nature of a model along with the emphasis on the communication patterns on a single host allows us to focus on the benefits of offloading protocol processing specifically as a measure of overhead and gap.

3.2 EMO

We wanted a performance model that is not specific to any one protocol, but our choices were informed by our understanding of MPI over TCP over IP.

The Extensible Message-oriented Offload model (EMO) Figure 1. The latency and overhead that is necessary to communicate between components must include the movement of data when appropriate.

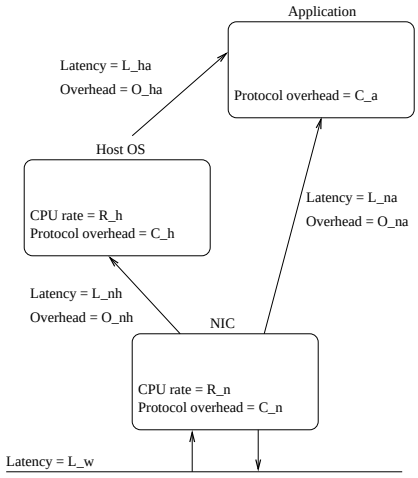

Figure 1. The Extensible Message-oriented Offload Model

The variables for this model are as follows:

- C_N = # cycles of protocol processing on NIC
- R_N = Rate of CPU on NIC in MHz
- L_{NH} = Time to move data and control from NIC to Host OS
- C_H = # cycles of protocol processing on Host
- R_H = Rate of CPU on Host in MHz
- L_W = Time to move data and control from network to App
- L_{HA} = Time to move data and control from Host to App
- L_{NA} = Time to move data and control from NIC to App
- C_A = # cycles of protocol processing at Application
- O_{NH} = # host cycles to move data and control from NIC to Host OS
- O_{HA} = # host cycles to move data and control from Host OS to App
- O_{NA} = # host cycles necessary to communicate and move data from NIC to Application

3.2.1 Extensibility

The model allows for extensibility with respect to protocol layers. We hope this model can be useful for researchers working on offloading parts of the MPI library (like MPI_MATCH) or parts of the matching mechanisms for any language or API. We constructed the model so that it can grow through levels of protocols. For example, our model can by extended, or telescoped, to include offloading portions of MPI. We simply add the following variables:

- C_A1 = # cycles of protocol processing at Application level 1
- L_{AA1} = Time to move data and control from App to App level 1
- O_{AA1} = # host cycles to move data and control from App to App level 1

These variables are included in the overhead and latency equations. The application becomes the MPI library and application-level 1 becomes the user level application. If there is another library or intermediate data handler like a LINDA library, another level of application, application-level 2, is introduced.

3.2.2 Overhead

EMO allows us to explore the fundamental cost of any protocol, its overhead. Overhead occurs at the per-message and per-byte level. Our model allows us to estimate and graphically represent our understanding about overhead for various levels of protocol offload.

Overhead is modeled as

$$\text{Overhead} = O_{NH} + C_H + O_{HA} + C_A + O_{NA}$$

. However, all methods will only use some of the communication patterns to process the protocol. Traditional overhead, for example, will not use the communication path between the NIC and the application and does no processing at the application.

$$\text{Traditional\_Overhead} = O_{NH} + C_H + O_{HA}$$

3.2.3 Gap

Gap is the interarrival time of messages to an application on a receive and the interdeparture time of message from an application on a send. It is a measure of how well-pipelined the network protocol stack is. In a well-pipelined protocol stack, gap is also a measure of how well-balanced the system is. If the host processor is processing packets for a receive very quickly, but the NIC cannot keep up, the host processor will starve and the gap will increase. If the host processor is not able process packets quickly enough on a receive, the NIC will starve and the gap will increase. If the network is slow, both the NIC and host will starve. Gap is a measure of how well-pipelined the protocol stack is and in the limit, gap is a measure of which is the bottleneck, the host CPU, the NIC CPU or the wire.

In the extreme, if the protocol stack is not well-pipelined, only one message may be processed at a time. If we assume that X is the size of the message, then the upper limit on gap is

$$\text{Gap} = \frac{\frac{C_N}{R_N}}{X} + \frac{\frac{C_H}{R_H}}{X} + \frac{L_W}{X}$$

On the other hand, if the protocol stack is well-pipelined and buffers are full, the gap is a measure of the slowest part of the packet processing. The lower limit on gap is

$$\text{Gap} = max(\frac{\frac{C_N}{R_N}}{X}, \frac{\frac{C_H}{R_H}}{X}, \frac{L_W}{X})$$

In fact, the speed of the host CPU, the NIC CPU or the network can be modified as the system is developed to create a more balanced system and further lower the lower limit on the gap for a protocol implementation.

3.2.4 Latency

Latency is modeled as

$$\text{Latency} = \frac{C_N}{R_N} + L_{NH} + \frac{C_H}{R_H} + L_{HA} + L_{NA} + \frac{C_A}{R_H} + L_W$$

However, all methods will only use some of the communication patterns to process the protocol. Traditional network protocols, for example, will not use the communication path between the NIC and the application and does no processing at the application.

$$\text{Traditional\_Latency} = \frac{C_N}{R_N} + L_{NH} + \frac{C_H}{R_H} + L_{HA}$$

3.3 Mapping EMO onto LAWS

EMO can be mapped directly onto LAWS which is useful in order to provide a context for the model in the larger offload community. Because LAWS concentrates on an arbitrary number of bytes in a specified amount of time and EMO concentrates on an arbitrary amount of time for a specified amount of bytes, we will have to make a few assumption.

The parameters that make up the ratios in the LAWS model are below. Please see [11] for a complete description.

- o - CPU occupancy for communication overhead per unit of bandwidth

- a - CPU occupancy for the application per unit of bandwidth
- X - Occupancy scale factor for host processing
- Y - Occupancy scale factor for NIC processing
- p - Portion of communication overhead o offloaded to NIC
- B - Bandwidth of network path

LAWS assumes a fixed amount of time. Let the fixed amount of time be equal to the time to receive a message of length N on a host. We'll call this time T. This allows us to determine the total number of host cycles possible.

$$C_t = R_h * T$$

For simplicity let's define an overhead total for EMO.

$$O_T = C_N + O_{NH} + C_H + O_{HA} + C_A + O_{NA}$$

Now that we have a fixed time T, a fixed number of bytes N, and the total number of host cycles C_t, we can map EMO onto the LAWS parameters.

The most difficult part of the mapping from EMO to LAWS is the fact that the communication overhead o is constant while the percentage offloaded p is variable. Thus, p is a ratio used to compare two different offload schemes. Our offload schemes are modeled with different values for C_H and C_N to reflect this difference. We use C'_H to represent the amount of protocol processing done on the NIC for a second offload scheme. We assume that C_N is incremented by $C_H - C'_H$ since this is the assumption of the LAWS model. Changes to the actual amount of protocol processing under various offload schemes are reflected in the LAWS model ratio β. Changes to the amount of protocol processing done under various offload schemes are reflected directly in different values in EMO.

$$\begin{aligned} o &= O_{NH} + C_H + O_{HA} + C_A + O_{NA} \\ a &= C_T - o \\ X &= R_H \\ Y &= R_N \\ p &= \frac{C_H - C'_H}{o} \\ B &= \frac{N}{T} \end{aligned}$$

LAWS derives all of its ratios from these parameters with the exception of β. The structural ratio describes the amount of processing saved by using a particular offload scheme. We can quantify this directly from our model assuming the second offload mechanism is denoted by variables with '.

$$\beta = \frac{C'_N + O'_{NH} + C'_H + O'_{HA} + C_A + O'_{NA}}{C_N + O_{NH} + C_H + O_{HA} + C_A + O_{NA}}$$

Now we have all of the necessary elements to map EMO onto LAWS. This is useful for understanding how the EMO model fits into the larger area of offloading of protocol processing in commodity applications. We created EMO for high-end computing so we can explore gap and overhead for a message-oriented applications.

3.4 Mapping EMO onto LogP

The mapping from EMO to LogP is trivial. In fact, EMO can be seen as a submodel of the LogP model although we use it separately and for a different purpose. There is no mapping for the number of processors, P, in EMO. Otherwise, we assume a well-pipelined system and let X =the size of the message.

$$\begin{aligned} L = \text{Latency} &= \frac{C_N}{R_N} + L_{NH} + \frac{C_H}{R_H} + L_{HA} + L_{NA} + \frac{C_A}{R_H} + L_W \\ o = \text{Overhead} &= O_{NH} + C_H + O_{HA} + C_A + O_{NA} \\ g = \text{Gap} &= max(\frac{\frac{C_N}{R_N}}{X}, \frac{\frac{C_H}{R_H}}{X}, \frac{L_W}{X}) \end{aligned}$$

4 Using EMO - A Case Study

While we have presented results that validate the EMO model elsewhere[7], it is also essential that EMO can be used as a tool either for developing a new protocol or for determining the appropriate design for offloading an existing protocol. This case study demonstrates the usefulness of EMO as a language for comparing and developing methods of offloading protocol processing.

The work we have done on offloading parts of the IP and TCP protocol onto a NIC[8, 4, 5, 6] and the experiences of our colleagues[9] has taught us that the keys to increasing performance is to reduce the communication costs between the network and the application and to reduce the number of cycles that the *data* must travel before being delivered to the application. Because we are only concerned about the number of cycles the data must travel, we process the protocol headers normally. We used EMO to create a method of bypassing most of the protocol processing of TCP called Splintered TCP.

First we will explain Splintered TCP using the same communication model we used to outline EMO in Figure 1. Next, we discuss interrupt coalescing, zero-copy TCP stacks and TCP offload engines in relation to splintered TCP. Finally, we will model the various methods for overhead and latency using EMO.

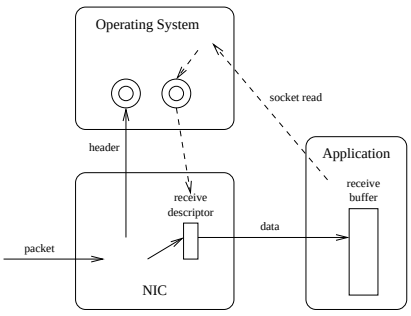

Figure 2. The Splintered TCP Architecture

4.1 Splintering TCP

Figure 2 presents a graphical illustration of our approach to splintering the processing associated with the TCP protocol. In this case we only illustrate the processing done while receiving datagrams. In this illustration, solid lines indicate the paths taken by datagrams, while dashed lines represent control activities.

We start by considering the control path. In particular, we assume that the application issues a non-blocking socket read before the data has arrived, i.e., a "pre-posted" read. Eventually, after traversing several layers of libraries, this read is transformed into a request that is passed to the operating system. The OS can then build a descriptor for the application buffer. This descriptor includes physical page addresses for the buffer. Moreover, as the OS builds this descriptor, it can easily ensure that these pages remain in memory (i.e., they will be pinned) as long as the descriptor is active in the NIC.

Now, we consider the handling of datagrams. When a datagram arrives, the NIC first checks to see if the incoming datagram is associated with a descriptor previously provided by the OS. If it finds such a descriptor, the NIC will DMA the data portion of the datagram directly to the application buffer, providing a true zero copy, and make the header available to the OS. If the NIC does not find the needed descriptor, it simply makes the entire datagram available to the OS for "normal" processing.

Perhaps more interesting than the functionality that we intend to put on the NIC is the functionality that we plan to leave in the OS. As we have described, we leave memory management in the OS and only provide the NIC with the mapping information that it needs to move data between the network and the application. We also plan to leave all failure detection and recovery in the OS.

Where is the acknowledgment generated? This is an open question and depends on the needs of the application. For synchronization messages, applications may choose to mark a receive so that an acknowledgment is generated directly on the NIC. For long messages in which there is no other check for data integrity, the application (or application library) may want to generate the acknowledgment through a system call to the operating system. If the checksum is generated and checked on the NIC, the operating system may also acknowledge the data. If the application or the operating system acknowledge the data, the round-trip time estimation on the sender side may be greatly affected and the TCP stacks may need to adjust the parameters of their congestion avoidance algorithms in order to avoid to much dependence on the measured round-trip time.

Successful offloading of congestion control along with direct DMA from the NIC to the application should substantially decrease the amount of CPU overhead associated with communication. Additionally, splintering allows the operating system to maintain appropriate control over resource management of the host processor and the NIC. For a more complete description of Splintered TCP, see [5].

4.2 Interrupt Coalescing

Interrupt coalescing is an algorithm in which the NIC waits for a certain number of interrupts or until a timeout threshold before interrupting the host processor. This allows the system to mitigate the cost of an interrupt by spreading it over many messages. This reduces overhead especially when the network frame size is small like the Ethernet. In addition to the overhead savings of interrupt coalescing, its other advantage is that it is widely available on NICs and it is cheap to implement since there is little logic on the NIC and very little state.

4.3 TCP Offload Engines

Since offloading a small part of the TCP/IP protocol onto a NIC is so successful, we naturally wonder whether offloading *more* of the TCP/IP stack would prove even more beneficial. In fact, the trend has been toward offloading all of the TCP/IP stack onto the NIC.

However, there are serious limitations to offloading the entire stack. Applications can no longer choose which protocol to use (TCP or UDP) as only TCP gives any performance advantages. If the connection management is offloaded, the amount of memory necessary to use such NICs in a large cluster would rival the host's memory capacity. Power consumption has become a very real part of the cost of a system and the hardware and power necessary for such sophisticated NICs can become prohibitive. Finally, creating a full TOE in hardware with enough memory to be scalable is very expensive.

Splintered TCP is an attempt to gain the performance advantages of TOEs at a fraction of the cost by allowing offloading onto a commodity NIC. The commodity NIC

would need a small amount of logic and a small amount of memory, but not much more than a commodity NIC contains today. For a full description of Splintered TCP memory constraints see [5].

4.4 Zero-copy

Zero-copy techniques created to bypass the memory copy are commonly used to decrease per-byte overhead. These techniques must also offload checksumming in order to gain the overhead savings. The performance measurements provided by [2] claim an overhead savings of up to 70% using zero-copy techniques to reduce per-byte overhead and large maximum transmission units (MTUs) to reduce per-packet overhead.

'Zero-copy' refers to two distinct ideas. The first is eliminating copying of network data in the IP stack on the host. This is an important improvement and has significantly improved IP stack performance over the past several years. The second is the idea of eliminating all copying of network data, including the final copy from kernel memory to user memory. Although this technique has been demonstrated successfully in some cases, zero-copy *to* user space has yet to be proven generally useful in an operating system standard release.

Implementations of zero-copy TCP are similar to splintered TCP in that it separates data and headers. The overhead associated with the memory copy is reduced and, because most common zero-copy TCP implementations are implemented using Myrinet [2], a large MTU mitigates interrupt overhead. However, the protocol headers are still processed before the application has access to the data and the latency of a message remains high.

4.5 Overhead using EMO

Figure 3 graphically represents the protocol processing overhead of some methods for decreasing protocol processing overhead. We compare interrupt coalescing, TCP offload, traditional zero-copy TCP, and splintered TCP.

Interrupt coalescing amortizes the cost of the O_{NH} over many messages. In order to model advantages of interrupt coalescing in EMO, we measure the overhead in the limit as O_{NH} approaches zero.

$$\text{Interrupt\_coalescing\_Overhead} = C_H + O_{HA}$$

Interrupt coalescing still requires the copy between the operating system and the application and so overhead is still linear in the size of the message.

Zero-copy techniques generally use fbufs[3] or page pinning to achieve memory protection without the price of a memory copy. If we do not consider the cost of the memory protection mechanisms the overhead removes the linear

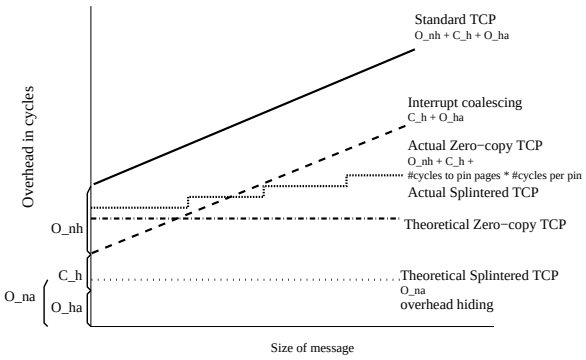

Figure 3. Extensible Message-Oriented Offload - overhead

dependence on message size in O_{HA}. Because the memory copy (and not the scheduling and context switch) is the limiting factor in O_{HA}, we consider the overhead as O_{HA} approaches zero.

$$\text{Zero-copy\_Overhead} = O_{NH} + C_H$$

However we do need to consider the cost of pinning pages because it is not trivial. The cost to pin a page is C_{pin} and the size of a page is $Size_{page}$.

$$\text{Zero-copy\_Overhead} = O_{NH} + C_H + C_{pin} * \frac{X}{Size_{page}}$$
$$X = \text{size of message}$$

This decreases the dependence of overhead on the size of the message, but the total protocol processing overhead is still tied to the per-byte overhead.

Splintered TCP uses the same memory protection methods as the traditional zero-copy TCP stacks. The difference between Splintered TCP and zero-copy TCP methods is that the event notification between the NIC and the host happens between the NIC and the application rather than the NIC and the operating system. However, the overhead between is the same ($O_{NH} = O_{NA}$) so the full overhead of Splintered TCP is the same as with zero-copy techniques.

$$\text{Splintered\_TCP\_Overhead} = O_{NA} + C_H + C_{pin} * \frac{X}{Size_{page}}$$

The major advantage of Splintered TCP, however, is that the overhead incurred by protocol processing does not have to be paid up front as it does in zero-copy TCP methods. This

process is similar to latency hiding, so we call it overhead hiding. Because we pin the pages during the pre-posted receive, we pay this overhead during the library call to set up the message receive descriptor. We notify the application when the data arrives rather than the operating system and we push the header into the operating system, but allow the application to determine when these headers are processed. The data bypasses the protocol processing. The application has the flexibility to determine when protocol processing occurs.

TCP Offload Engines and iWARP (TCP over RDMA-enabled NICs) are able to reduce the overhead of protocol processing by offloading all processing onto the NIC. The overhead still associated with TOE and iWARP is O_{NA}. Some TOEs and RDMA-enabled NICs (RNICs) claim to perform memory management and so are able to offload the overhead associated with memory protection. However, current versions of TOEs must still pay for memory protection. TOE and iWARP, through the Sockets Direct Protocol (SDP) suggest using full TCP offload only for large messages because of the overhead associated with current memory protection mechanisms. The TOE overhead follows.

$$\text{TOE\_Overhead} = O_{NA} + C_{pin} * \frac{X}{Size_{page}}$$

$$X = \text{size of message}$$

In this case, the overhead of TOE is similar to the overhead of Splintered TCP and zero-copy TCP mechanisms, but the per-message processing occurs on the TOE or NIC.

Zero-copy techniques decrease overhead over regular TCP stacks by removing O_{HA}, but they do add the overhead of memory management techniques. Splintered TCP has the same overhead as zero-copy techniques, but provides the application the flexibility to hide some of this overhead. TOEs and RNICs provide the most reduction in overhead, to either

$$O_{NA} + C_{pin} * \frac{X}{Size_{page}}$$

or with extra hardware functionality to

$$O_{NA}$$

What we can tell from our conceptual graphic of the overhead of each method is that the advantages of Splintered TCP or TOEs is dependent on the overhead cost of of O_{NA}. If the overhead of moving from the NIC to the application is very low, the offloading techniques will be useful. If, however, the cost is high, the techniques will not yield better results. Finally, we see graphically the intuitive understanding that offloading techniques will provide the most improved performance as the size of the message increases.

We assume that $C_H < O_{NH}$. We assume that the cost of traveling through the OS is less than the cost of interrupting the host processor. The cost of processing an interrupt may be smaller than the cost of processing a message, but this simply shifts the position of the overhead lines, it does not affect their slope. Finally, we assume that O_{HA} is linear with the size of the message. For small messages, caching will flatten this line. For large messages, the assumption still holds.

4.6 Latency

Figure 4 graphically represents the latency of some methods for decreasing protocol processing overhead. We compare interrupt coalescing, TCP offload, traditional zero-copy TCP, and splintered TCP.

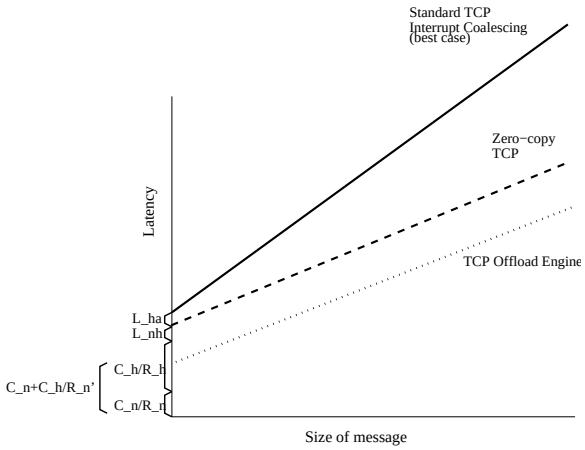

Figure 4. Extensible Message-Oriented Offload - Latency

Interrupt Coalescing decreases the overhead by waiting to interrupt the operating system until a number of messages have arrived for processing. While this decreases the amount of protocol processing overhead, it actually increases latency. We have graphically represented the best case latency for the interrupt coalescing method. This is the same as the latency of a message processed using a traditional TCP stack.

Zero-copy techniques should decrease the latency of a message since they remove the copy and subsequent event notification between the operating system and the application. However, the time to pin a page is often large enough so that it is actually faster to perform the memory copy and the context switch for small messages rather than pin the

page.

$$\text{Zero-copy\_Latency} = \frac{C_N}{R_N} + L_{NA} + \frac{C_H}{R_H} + \frac{C_{pin} * \frac{X}{Size_{page}}}{R_H}$$

$$X = \text{size of message}$$

This latency will increase as X increases. The slope of the increase in latency as the size of the message increases is not as steep in zero-copy TCP as it is in standard or interrupt coalesced TCP. This is because while latency is still influenced by L_{NA} which is linear as the size of the message increases, it is no longer affected by L_{HA} which is also linear as the size of the message increases.

TOEs and iWARP using RNICs process the protocols on NIC and so the speed of the NIC becomes much more important.

$$\text{TOE\_Latency} = \frac{C_H + C_N}{R_N} + L_{NA}$$

Latency should be sufficiently reduced when using ASICs which are very fast. Also, if the memory protection mechanisms are on the NIC, then the memory protection mechanisms are faster as well. Current TOEs and iWARP mechanisms do not generally decrease the latency of very small messages because the page-pinning is still done on the host.

Because of overhead hiding, the latency of a message in Splintered TCP is low even in for small messages. The page is pinned before the message is received so the latency of pinning the page is not incurred in the message receive path. An optimization for small messages is to simply DMA the data of small messages into the event queue of the application. This removes the need to pin a separate page for small messages.

$$\text{Splintered\_TCP\_Latency} = \frac{C'_N}{R_N} + L_{NA}$$

Some additional processing will need to be performed on the NIC so C_N will be increased to C'_N. However, C'_N is much lower than the $C_N + C_H$ processing cycles necessary for TOEs.

We assume that $L_{NH} = L_{NA}$ although this is probably a conservative estimate because of the polling mechanisms for L_{NA}. We also assume that $L_{NA} = L_{HA}$. We assume that the DMA is equal to the burst transaction rate of the PCI bus and that memory copy is equal to $\frac{1}{2}$ of the memory bus. Finally, we assume for our system that the PCI bus is set to $\frac{1}{2}$ of the memory bus speed so $L_{NA} = L_{HA}$. This ratio of I/O bus to memory bus speed will need to be determined for each system and this ratio will need to incorporated into the model. However, the ratio will be constant on a system.

The EMO provides a graphical representation of how much latency savings the speed of the NIC and the number of cycles of processing offloaded will yield. This provides us with more information about the necessary processing power needed to provide latency and gap savings using Splintered TCP.

Also, EMO provides us with a language in which to compare latencies of various approaches to protocol development. For example, let's say we want to know how fast our NIC processor will need to be in order to be competitive with a TOE in terms of latency. We assume we are offloading a quarter of the protocol processing onto the NIC for our Splintered TCP. For the sake of convenience we assume that $C_H = 4 * C_N$. Both the amount of cycles offloaded for a Splintered TCP implementation and the ratio of cycles on the host to cycles on the NIC can be determined for a particular TCP stack. We call the speed of the NIC of the TOE engine R_{N1} and the speed of the NIC for the Splintered TCP solution R_{N2}.

$$\frac{C_H + C_N}{R_{N1}} = \frac{.25 * C_H + C_N}{R_{N2}}$$

$$\frac{5 * C_N}{R_{N1}} = \frac{2 * C_N}{R_{N2}}$$

$$R_{N2} = \frac{2}{5} * R_{N1}$$

In order for us to achieve latencies equal to the TOE, our Splintered TCP would need to be implemented on an FPGA or processor at least $\frac{2}{5}$ as fast as the TOE engine.

This validates our assumptions since Splintered TCP only offloads a quarter of the protocol processing from the host to the NIC and bypasses the further protocol processing assuming it will be finished by the host at a later time. On the other hand, the TOE offloads all of the host protocol processing onto the NIC and assumes this processing must complete before data is delivered to the application. Therefore the TOE would require a faster NIC.

4.7 Summary

Splintering attempts to more efficiently use the operating system to control communication. By moving select functions of communication onto the NIC, we can decrease the number of interrupts to the operating system while still allowing the operating system to manage resources.

More specifically, splintering removes the processing of the protocol headers from the path of the data whenever possible. By moving the data to the application immediately and holding the headers for later processing we can divorce the management of the network from the management of the data. If the application can access the data as early as possible, it can effectively bypass the protocol processing and remove it from the latency path of the message.

If the application can control when to process the protocol headers, it can potentially hide the overhead of the protocol processing.

We used EMO to compare splintering with interrupt coalescing, zero-copy TCP and full TCP offload. The model demonstrates overhead hiding and the substantial decrease in latency for the TCP offload and Splintered TCP method. Furthermore, we can see the usefulness of EMO in determining hardware and software implementations given certain design parameters (as in latency guidelines).

5 Conclusions and Future Work

The extensible message-oriented offload model (EMO) allows us to explore the space of network protocol implementation from the application messaging layers through to the NIC on a message by message basis. This new language gives us a fresh understanding of the role of offloading in terms of overhead, latency and gap in high-performance systems.

We mapped EMO onto existing performance models. In fact, EMO can be looked at as a refinement of the LogGP model. However, we think of this as a model for implementation of a language rather than algorithmic analysis. EMO provides a simpler and easier to understand language for protocol development than LAWS. Furthermore, EMO allows us to explore the tradeoffs of offloading in a message-oriented rather than stream-oriented environment.

EMO further motivated and illustrated the use of Splintered TCP to create a method of protocol bypass which significantly reduces latency and allows the application control over protocol processing overhead. We were able to compare Splintered TCP to other methods of offloading to minimize overhead.

We plan on exploring the model of gap in EMO to bound the resource requirements for NICs or TCP offload engines at 10Gb/s and 40Gb/s speeds. We plan also to extend EMO to include memory management considerations such as caching.

We are currently completing work on validating EMO. Additionally, we are modeling, validating and quantifying the various memory protection mechanisms used in zero-copy network protocols.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[2] J. Chase, A. Gallatin, and K. Yocum. End-system optimizations for highspeed TCP. In *IEEE Communications, special issue on TCP Performance in Future Networking Environments*, volume 39, page 8, 2000.

[3] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.

[4] P. Gilfeather and A. Maccabe. Making TCP viable as a high performance computing protocol. In *Proc. of the 3rd Annual Symposium of the Los Alamos Computer Science Institute*, October 2002.

[5] P. Gilfeather and A. Maccabe. Splintering TCP. In *Proc. of the 17th International Symposium on Computer and Information Sciences*, October 2002.

[6] P. Gilfeather and A. B. Maccabe. Connection-less TCP. In *Workshop on Communication Architecture for Clusters*, April 2005.

[7] P. Gilfeather and A. B. Maccabe. An extensible message-oriented offload model for high-performance applications. Technical Report TR-CS-2005-28, University of New Mexico, 2005.

[8] P. Gilfeather and T. Underwood. Fragmentation and high performance ip. In *Proc. of the 15th International Parallel and Distributed Processing Symposium*, April 2001.

[9] A. B. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience in offloading protocol processing to a programmable nic. In *Proceedings of IEEE International Conference on Cluster Computing*, September 2002.

[10] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 85–97, Denver, Colorado, June 1997.

[11] P. Shivam and J. Chase. On the elusive benefits of protocol offload. In *SIGCOMM workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI)*, August 2003.