Splintering: A Technique to Increase Performance in Commodity TCP/IP*

Patricia Gilfeather
Scalable Systems Lab
Department of Computer Science
University of New Mexico
pfeather@cs.unm.edu

*Los Alamos Computer Science Institute SC R71700H-29200001 and Albuquerque High Performance Computing Center through IBM SUR

Contents

1 Introduction

Clusters built from commodity hardware and software have several advantages over more traditional supercomputers. Commodity clusters are cheap and ubiquitous, and so they are easier to design, program and maintain. However, as high-speed networks reach 10Gb/s speeds and modern computer architectures reach past 1GHz, new commodity clusters are unable to harness this power.

In addition to the well-known bottleneck of the memory copy from kernel to user space, two of the largest impediments to using commodity hardware and protocols are the interrupt pressures associated with Ethernet-sized frames and the storage and management of TCP connections in large systems.

One of the most effective tools for increasing the efficiency of high-performance computing has been the use of intelligent Network Interface Cards (NICs). Traditionally, offloading work onto the NIC has been an all or nothing proposition. For example, interrupt coalescing, a method which interrupts the operating system after a number of interrupts have been accumulated, offloads none of the original processing work associated with the communication stack. Instead, it adds the extra work of counting interrupts at the NIC. On the other hand, the newest research considers offloading the entire TCP/IP communication stack onto the NIC[33, 32]. These extremes both come with their drawbacks.

In order to take full advantage of offloading, we must carefully determine which functionality in the communication protocol stack will give us the most benefits if offloaded. As we will illustrate, some functions, like fragmentation and defragmentation or IP checksum, can be offloaded with positive results. However, other functions, like TCP checksum, offer little gain. I propose studying *splintering* a commodity protocol stack to improve TCP performance. Splintering is the process of determining which functionality to extract from the protocol stack and extracting it. *By splintering the functionality of the TCP/IP stack, we retain the advantages of commodity protocols and gain the performance efficiencies of appropriate offloading onto peripheral devices making communication in high-performance computing faster and more flexible.*

First, this paper will outline the general problem space. We will investigate the performance metrics of high-performance computing and study the advantages and the performance challenges with respect to commodity hardware, operating systems and protocols. After discussing the problem space, we will discuss each of the major performance bottlenecks in detail. We make no research proposals in the area of relieving the memory copy bottleneck and so we will only review the relevant literature. We have completed work on the process of splintering and offloading IP fragmentation and defragmentation as a means of relieving interrupt pressure. Finally, we propose to splinter and offload TCP congestion control as a means of relieving connection setup and management inefficiencies.

2 The Problem Space

The problem space associated with this research is bounded by the special performance limitations of commodity components and the characteristics and needs of high-performance computing. We will first introduce the major performance bottlenecks of TCP/IP in high-performance computing, then we will discuss the characteristics that are specific to high-performance computing. We will then propose a set of performance metrics that reflect the advantages of commodity protocols and the needs of the high-performance computing community. Finally, we will specifically compare commodity and specialized approaches to various levels of computing.

2.1 The Performance Bottlenecks

Research in commodity protocols is most often associated with the Wide Area Network (WAN) world. However, the needs and patterns of the high-performance computing community are quite different. We will first discuss the performance bottleneck that is most often associated with high-performance computing, the memory copy. Next, we will discuss the two most serious bottlenecks in high-performance computing, processor overhead and scalability.

The bottleneck which is most often associated with performance problems in commodity protocols is not primary in today's systems. Historically, because memory bus speeds were slow, the copy necessary to move a message from kernel space to user space in a put protocol like TCP was considered prohibitively expensive for high-performance computing models [16, 26, 27]. Stacks have become more sophisticated in recent years and have eliminated all but this single copy. Debate continues as to whether the elimination of this final copy will actually increase performance[30]. Many researchers have created sophisticated implementations of the TCP/IP stack to remove all memory copies and put message data directly into application space[7, 8]. None of these current implementations, however, take advantage of commodity hardware so they remain quite expensive.

On commodity hardware, 100 Mb/s Ethernet adapters are capable of near line-rate TCP/IP performance (90% of available bandwidth) with low CPU overhead (15-20%) where CPU overhead is the amount of CPU used in communication. Projecting this performance out to Gigabit speeds, we can expect to see TCP/IP performance drop as we reach near 100% CPU overhead. Our experiments confirm that a 933MHz Pentium with a Gigabit network achieved peak bandwidth of 820 Mb/s (80% of the available bandwidth) at full CPU overhead (100%) [10]. Bottlenecks to high speed performance no longer reside with the network infrastructure. With modern commodity hardware supporting memory-to-IO bandwidths of 480MB/s (3840 Mb/s), memory performance is not the limiting factor. In fact, interrupt handling and the resulting CPU overhead are the major performance issues in efficient high-speed networks. We propose a solution to the interrupt pressure bottleneck and empirically show its dramatic success at reducing CPU overhead.

When considering performance with respect to cluster technology, however, we must be aware of the orthogonal performance metric, scalability. In addition to interrupt pressure, the cost of setting up and maintaining connections in a connection-oriented protocol like TCP makes larger clusters of nodes inefficient. The congestion control mechanism in TCP requires that connections ramp up to full bandwidth and this slow start makes connection setup expensive, especially for shorter messages[3]. Maintaining an open connection, on the other hand, requires maintaining several state variables per connection. We plan to specify a solution to the slow start-up of TCP connections and offload the congestion control mechanism associated with connections in order to show that commodity protocols can be competitive even in the arena of large clusters.

There have been several moderately successful attempts at addressing the new performance bottlenecks in high-performance local area network computing. The first, most-obvious choice, is to use protocols more suited to high-speed networking, and there are several well-established protocols in the research community. Namely, protocols like FM or Portals[21, 25]. However, since IP remains the most common protocol within LAN applications and certainly in the WAN world, IP implementations are widespread and therefore inherently appealing. Changing protocols means maintaining protocol stacks separately for the WAN (which remains TCP/IP) and the LAN and also requires a great deal of re-engineering. We sought to implement a scalable and efficient way of addressing the major bottlenecks in high-speed networks, while still maintaining all of the advantages of a commodity protocol.

2.2 Characteristics of High-Performance Computing

Unlike the WAN world, high-performance computing is generally characterized by the reliability of the message-passing medium and the intensity of the computational needs of the application. Additionally, programming models and communication libraries manage network traffic patterns and generally use only very small and very large messages[24, 17]. Reliability, computational needs, traffic management and specific message sizes are the characteristics of high-performance computing that will bound how we look at commodity protocols.

Currently, the most scalable approach to maintaining a reliable method for sharing data is message-passing over a a system area network (SAN). In practice, environments for high-performance computing have allocated enough resources to maintain a network with little congestion and SAN routing information is likely to be static and cached[18]. Generally, the topology of the network is tree-like. In addition to the reliable wire, high-performance networks will few messages due to congestion at an intermediate source for three reasons. First, enough resources have been allocated for the purpose of routing. Second, routing information is static, and third, applications or libraries manage messaging traffic.

In addition to reliability, high-performance computing assumes that the application will need and exercise control over sharing of data. Applications, or the parallel message passing libraries that they employ (e.g. MPI), share data in models (like RTS/CTS) which take into account the amount of buffer space at the receiver that is available for new data[18]. In other words, applications keep message passing from creating congestion at the endpoints by managing flow control. Additionally, applications will attempt to avoid intermediate congestion by being aware of the network topology and employing patterns of communication that exploit the topology to decrease the amount of contention on the network (e.g. nearest-neighbor message passing).

While the wide area network world is often characterized by small messages and bursty traffic, the high-performance computing environment uses both small, highly-synchronized messages and very large messages. There is an inherent burstiness to some commodity protocols like TCP that can be problematic in the area of multi-media and real-time applications that rely on a streaming model of computation[3, 23]. High-performance applications rarely consider data as streaming data and are more likely to consider data as discrete and bounded which fits well with the model of communication employed by most commodity protocols (like IP).

Finally and most importantly, 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.[17]

High-performance computing is characterized by reliability and performance. The message-passing medium and the resources allocated to routing alleviates congestion in the network and application flow control alleviates congestion at the endpoints. All of these characteristics will have an impact on how we look at commodity hardware and software. We will need to allow applications more control over message passing and the overhead of commodity protocols must be minimized.

The characteristics of high-performance computing particular to this study are as follows:

- High reliability,
- Application-level flow control,
- Application-level congestion avoidance,
- Very small and very large (bounded) messages, and
- Sensitivity to communication overhead

2.3 Performance Metrics for High-Performance Computing

In an effort to evaluate optimizations to commodity protocols, we will formalize the performance metrics for high-performance computing. The metrics we will evaluate are effective utilization, flexibility, scalability, interoperability and cost.

The primary performance characteristic we must address is lowering communication overhead on the host processor as high-performance computing is linearly sensitive to increases in CPU overhead[17]. Traditionally, however, bandwidth has been the measure of success in networking and we must certainly maintain competitive bandwidth since decreasing communication overhead by simply decreasing the amount of communication does not reflect a systemic improvement. A true measure of this performance is *effective utilization* which is calculated by dividing bandwidth by CPU overhead. We hope to achieve a high effective utilization which means that we are gaining the most bandwidth from the least amount of CPU overhead. As we lower communication overhead and maintain high bandwidth, our effective utilization will increase.

Any method for relieving performance bottlenecks must also allow for application-level flexibility of protocols, messaging styles or performance needs. Some applications need consistent latency (low jitter) for synchronization. On the other hand, some applications will be less sensitive to latencies than to bulk data flow. Also, different high-performance applications may need different high-level protocols. MPI can be implemented on TCP whereas many multi-media applications use UDP. Applications and application programmers must have the flexibility to emphasize their different needs.

Any solution which increases performance must be able to scale as the performance of hardware increases. We call this performance metric performance scalability. As the CPU, memory bus and network speeds continue to become faster, any solution to a performance bottleneck can only be considered a solution if the same bottleneck does not reappear.

Clusters come in many sizes. One of the important metrics by which to measure any changes to a protocol or implementation is its system scalability. A truly useful protocol will maintain performance regardless of the number of endpoints in the network. On the other hand, many proposed solutions are willing to sacrifice scalability for other performance gains.

Additionally, we must be aware of the performance advantages associated with commodity protocols like IP and TCP. By using commodity protocols we expect to increase our interoperability. Interoperability is not a binary metric. The high-performance computing community may choose a protocol that is fully compliant with respect to IP and can take advantage of IP routing, but which is only partially interoperable with respect to TCP. For example, such a protocol might take advantage of TCP's reliability without implementing the congestion control necessary for protocol completeness. Researchers may choose a trade-off with respect to compliance, completeness or global interoperability.

Practically, the most important performance metric in industry is resource usage. Here we want to decrease the amount of time needed to perform large calculations, but we also want to *minimize cost*. Commodity protocols decrease resource consumption in terms of power, space, hardware, software, training, programming and maintenance. As we make trade-offs for better performance in other areas, we may lose performance with respect to cost.

2.4 The Advantages of Commodity Components

It is often in the second group of performance metrics that we gain from the commodity approach. Commodity-based hardware and protocols provide an extensively tested, extensively developed, highly interoperable, and inexpensive alternative to specialized solutions. The cost advantages for the commodity approach reaches far beyond the price of hardware. Often code has already been created in the community so there is little to no development cost. In the remainder of this section, we discuss the advantages of commodity-based hardware, protocols and operating systems.

2.4.1 Hardware

Fast Ethernet is a great example of the trade-offs between inexpensive commodity and more expensive specialized hardware. While Fast Ethernet is very inexpensive, it is capable of only 1/10th the bandwidth of Myrinet[5]. In 1999, Gigabit Ethernet had comparable bandwidth with Myrinet, but its prices were high. However, in the last year, the price of Gigabit Ethernet NICs has dropped from around

$1000/NIC to $200/NIC[9, 11]. This is the advantage of commodity-based hardware. Gigabit Ethernet and Myrinet are competitors. While Myrinet has out-paced Gigabit Ethernet in performance, its cost is still prohibitively high for all but a few installations.

2.4.2 Network Layer Protocol

Internet Protocol (IP) is the ubiquitous network layer protocol. IP routing is absolutely necessary to remain interoperable in wide area networks. Additionally, IP routers remain the most cost-effective hardware choice and the IP routing mechanism is well-tested.

There are disadvantages to IP as well. IP is an old protocol and some of its assumptions are no longer valid. Generally, IP's hierarchical, dynamic routing is inefficient although LANs can maintain route table information and alleviate this weakness. Additionally, IP checksums reside in the header rather than at the end of the packet so computing the checksum requires maintaining additional state[28]. Also, headers are of various size requiring a check of the length field of each header.

2.4.3 Transport Layer Protocols

Transport Control Protocol (TCP) and its unreliable cousin User Datagram Protocol (UDP) are the most common transport layer protocols used today. Generally, TCP gives all of the advantages of a commodity protocol. Specifically, the commodity protocol approach is transparent to the layer above it which is the application in this case. In other words, hundreds of applications are written based on the TCP protocol. Interoperability would be severely limited without TCP. This interoperability has an added benefit. System dynamics are often hard to trace, but are very important. When two implementations of the TCP congestion control were both competing for bandwidth, the mismatch between the cooperative Reno that decreased its congestion window even on duplicate acks, and the ubiquitous Tahoe which does not use fast retransmit mechanism and therefore only reduces its window on timeout became very quickly apparent [29]. In a smaller environment, the incompatibility would have been much harder to find.

On the other hand, TCP contains a large amount of communication processing overhead to administer flow control, error discovery and correction, and to maintain connections. Many of these services are unnecessary to the high-speed network application. These networks are highly reliable with little to no errors so the error detection and recovery would be less intrusive if it were not on the main path for all messages. Congestion control is maintained by the application so it's existence in the transport layer is redundant. Some modifications to TCP have been introduced to alleviate the performance bottlenecks. These include larger window sizes which are necessary when trying to keep a fast pipe full, the option to acknowledge data selectively to avoid the go-back-n error recovery, and timestamps for rate detection[12]. These improvements have helped, but the connection-oriented nature of the protocol makes the ability to communicate with thousands of nodes highly inefficient and so the protocol is not scalable. This is the true expense of TCP.

2.4.4 Operating System

Generally, an *open source* operating system presents many advantages in an environment like the high-performance computing world. As a commodity, the operating system can be seen as well-tested and well-developed with an environment conducive to both application programmers and system administrators. Also, a commodity operating system means a lot of the basics of good system design are not yours to develop and test. Finally, the fewer additions or modifications to an operating system, the easier nodes are to update and administer.

The cost, however, is performance especially in scheduling and memory copying. An example of this is the disparity of the scheduler in 4.3 BSD Unix and the communication pattern in a loaded system. Since the scheduler favors short processes, compute-bound processes can starve[22]. Finally, the memory copy necessary between all existing operating systems and user space is based on the virtual memory system and can be expensive. While it is not the bottleneck, the memory copy contributes to the overall processing overhead.

2.5 Commodity versus Specialized

The advantages of the specialized approach to increasing performance in high-speed networking include much better use of the underlying hardware with lightweight protocols that take advantage of high-speed reliable networks. This gives specialized solutions like FM over Myrinet a distinct advantage with respect to CPU overhead since the excessive overhead associated with general protocols is gone[5, 21]. The additional CPU overhead savings come from offloading protocol processing onto the NIC. The disadvantages to the specialized approach are price, lack of interoperability, a less robust testing and development environment, and more sophisticated system administration.

The advantage of the commodity approach to increasing performance in high-speed networking is an inexpensive, well-tested, well-developed highly interoperable and robust system from hardware through application interface where many of the applications have already been developed and optimized. The disadvantage is the extra protocol processing which increases CPU overhead and latency and can decrease bandwidth (although generally, commodity networks of today are bound by CPU overhead rather than bandwidth). Additionally, more subtle disadvantages to the commodity approach are the strains within a community that contains researchers and users with very different needs. Can TCP, for example, satisfy the needs of both the mobile world as well as the well-connected reliable cluster world?

2.6 Summary

The high-performance computing environment provides a special set of characteristics and performance metrics. High-performance computing is generally very homogeneous. Networks in this environment tend to be very reliable and fast. Applications often provide their own sophisticated congestion control mechanisms and are most sensitive to processor overhead. In this environment, the metrics that are most indicative of performance are effective utilization (bandwidth/CPU utilization), flexibility, scalability, interoperability and cost. With this environment and these performance metrics in mind, we studied the advantages and disadvantages of commodity protocols, hardware and operating systems.

Can we get the advantages of both the commodity and specialized approaches? One way is to separate and isolate small parts of the functionality of a commodity and use specialized approaches to optimize this functionality. If we are careful in choosing what to separate, we can maintain the advantages of the commodity hardware, protocol and operating system while optimizing the function to gain the advantages of a specialized approach.

Now that we have defined the problem space, we will look at three of the performance bottlenecks of high-performance computing. We will begin with a short discussion of the memory copy bottleneck, then we will present completed research on solving the interrupt pressure bottleneck and we will complete with a proposal for solving the connection management bottleneck.

3 Removing Memory Copies

We will review why the memory copy is considered a bottleneck and why we believe that it is not critical in today's environments. Then, we will present a short discussion of solutions to the problem.

Most IP stacks still copy memory at least once: between user and kernel memory. If the bandwidth of the memory bus is inadequate, any memory copies that cannot be pipelined will have a direct impact on the communication bandwidth observed by the application. In addition, the copy between user and kernel memory contributes to the overall processing overhead.

When network speeds approach (or exceed) commodity memory bus bandwidth, the cost of copying network frames in memory, even once, becomes unacceptable. However, commodity memory-to-IO bandwidths are increasing and performance pressure from a single memory copy is becoming less important.

However, *even when memory copies do not directly impact the bandwidth seen by applications, they use resources (e.g., the memory bus and the PCI IO bus) which may also impact overall application and system performance, and they contribute to the overall processor overhead.*

Because memory copies are so often considered the most damaging performance bottleneck in commodity protocols and because so much work has been done in the past decade to alleviate this pressure, we will review the literature associated with reducing or eliminating memory copies. Certainly, reducing or eliminating the number of memory copies will decrease the overall processor overhead of communication.

3.1 Zero-copy

'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. The primary concern is that the overhead and special-casing in the page cache necessary to manage the transition between the two address spaces may exceed the overhead of a single memory copy into a user buffer.

3.2 Trapeze

The most well-known solution to full zero-copy TCP/IP is the Trapeze project[7] which offloaded part of the TCP/IP stack onto Myrinet cards and modified the FreeBSD kernel to create a zero-copy implementation. Since the original research, these modifications have been rolled into the released kernel. This approach uses a control program on the Myrinet NIC to separate TCP/IP headers and data and to perform checksums. The FreeBSD kernel processes the headers through the normal protocol and the data is moved to application space using fbufs.

The Trapeze implementation was very successful, achieving TCP/IP bandwidths of over 2Gb/s. This peak bandwidth uses Myrinet which allows for a variable MTU. In fact, peak bandwidths were measured at 8K MTU (the size of a VM page). Unfortunately, there are no CPU overhead numbers that correspond to this bandwidth. We can assume that since some or all of the TCP/IP stack has been offloaded, that CPU overhead should be low. Generally, zero-copy techniques will add the overhead of allocating or clearing buffers (fbufs) and so they are best used only for large messages[15]. Whether the latency for small messages remains low depends on whether an implementation only uses zero-copy on large messages.

The Trapeze project is not directly related to this study since the hardware used in the implementation is not commodity. The use of Myrinet cards and network means that this implementation is not generally interoperable or cost-effective. It does, however, point to the success of a zero-copy commodity protocol with respect to effective utilization. These results are summarized in Table 1.

Performance Metrics	Trapeze
increased effective utilization	good
application-level flexibility	fair
interoperability	*poor*
cost	*poor*
perf. scalability	fair
system scalability	unknown

Table 1. Performance parameters for Trapeze (using zero-copy on Myrinet)

4 Reducing Interrupt Pressures

We will first review the interrupt pressure bottleneck and show its primacy in high-performance networks of today, then we will describe the performance of two of the most well-known attempts at relieving this bottleneck. Finally, we discuss offloading communication work onto the NIC and present detailed performance characteristics of offloading a small function of the IP stack onto the NIC.

Interrupt pressure is the most significant concern for high-speed Ethernet networks, yet it is largely overlooked as a performance bottleneck. Table 2 presents the minimum inter-arrival times for 1500 byte packets at various network speeds. The minimum inter-arrival time is calculated by assuming a constant stream of 1500 byte packets. For example, the inter-arrival time for a 10Mb/s network is calculated as follows:

$$1500\text{bytes} \times 8\text{bits/byte} \times 1\mu\text{s}/10\text{bit} = 1200\mu\text{s}$$

If the network interface generates an interrupt for each 1500-byte packet, a processor with a 1Gb/s network interface should be capable of handling an interrupt every 12 μs. Although estimates of interrupt handling latency vary considerably and should be measured on each system, the PCI 2.1 specification estimates typical 33MHz latencies of 10μs to 20μs. Expected latencies on 66MHz buses should be

Network Speed	Inter-arrival time
10 Mb/s	1200 μs
100 Mb/s	120 μs
1 Gb/s	12 μs
10 Gb/s	1.2 μs

Table 2. Minimum Inter-arrival Times for 1500 byte Packets

lower. Even an extremely efficient operating system cannot hope to manage interrupts arriving at 12μs, and the situation becomes utterly hopeless for 10 Gb/s networks.

4.1 Interrupt Coalescing

We have several options for reducing interrupt pressures due to communication. Most conservatively, we can coalesce interrupts. The use of algorithms to reduce (or coalesce) the number of interrupts has been widespread. In this approach, the receiving NIC only interrupts the host after a specified amount of time or number of arriving packets.

In order to test the performance of interrupt coalescing, we used an unmodified copy of netperf 2.1pl3[19] to measure unidirectional UDP streaming bandwidth. We chose netperf as a testing platform because data from netperf are frequently cited by other high-performance projects. Additionally, among network performance measurement packages, netperf is one of few to have a reasonable UDP implementation. Netperf only reports the UDP bandwidth actually received, providing a reliable means of measuring bandwidth for an otherwise unreliable protocol. All reported measures are to within a 95% confidence interval.

We configured two 933 MHz, Linux 2.4.0(release) servers with version 0.49 of Jes Sorenson's Acenic driver (patched only to support tracedumps). The machines were connected by a cross-over fiber cable (with no switch).

The Acenic driver has four configurable parameters, the receive-side interrupt threshold, the send-side interrupt threshold, the receive-side maximum interrupt inter-arrival time and the send-side maximum interrupt inter-arrival time. The default parameters were used for the default interrupts. To disable interrupt coalescing, the receive-side interrupt threshold and the send-side interrupt threshold were set to

0 and the maximum interrupt inter-arrival times were set to 0. For maximum interrupts, the receive-side and sender-side interrupt thresholds were set to 1000 and the maximum interrupt inter-arrival times were set to 100μs.

As we see in Figure 1, while effective utilization increases slightly with interrupt coalescing, the amount is not significant. In fact, improper adjustment of the interrupt coalescing algorithm can have detrimental effects. Another significant disadvantage is that small packets of a latency-bound application are not allowed to move directly through the NIC to the IP stack on the host. Instead, they are subject to being coalesced and latency-bound applications can see their performance suffer. In other words, application-level flexibility is severely compromised.

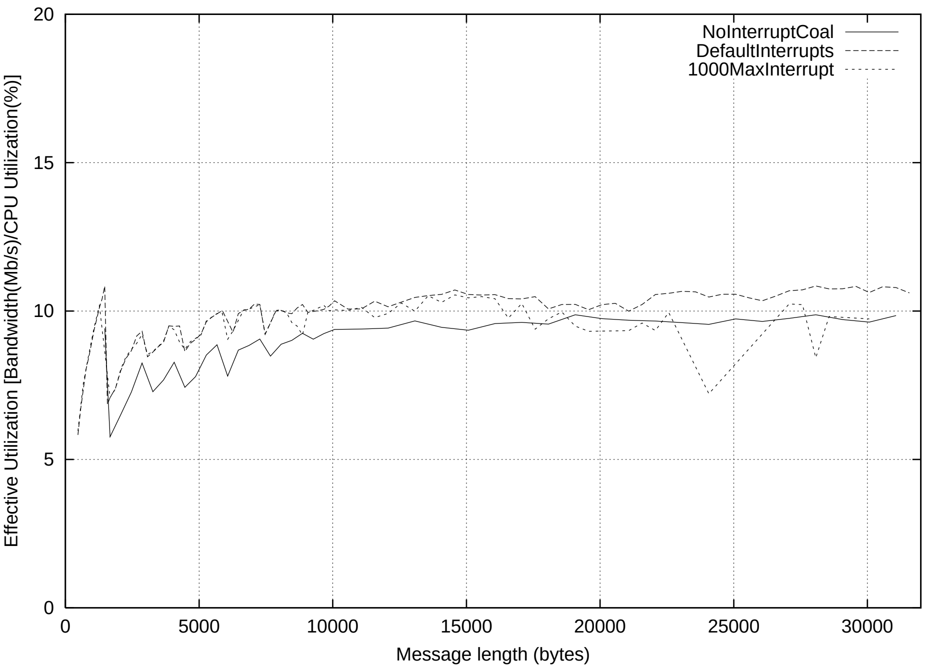

Figure 1. Effective utilization using no interrupt coalescing, default interrupt coalescing and extreme interrupt coalescing

Interrupt coalescing is scalable and interoperable. It solves the problem of interrupt pressure cleanly and elegantly. However, reducing the number of interrupts by holding messages for a short time does not significantly increase the effective utilization of the host processor. While this technique does maintain bandwidth and interoperability and is inexpensive, its greatest disadvantage is the high variance in latency which leads to a significant loss in flexibility of the application. These results are summarized in Table 3.

Performance Metrics	Interrupt Coalescing
increased effective utilization	fair
application-level flexibility	*poor*
interoperability	good
cost	good
perf. scalability	fair
sys. scalability	fair

Table 3. Performance parameters for Interrupt Coalescing

4.2 Jumbo Frames

Standard Ethernet Frames are 1518 bytes, including the MAC header and the CRC trailer. Jumbo frames, supported by many Gigabit NIC and switch manufacturers, increase the size of an Ethernet frame to 9018 bytes. This reduces the number of packets necessary to transport a large message and thereby reduces the number of interrupts at the receiving and transporting hosts.

To test jumbo frames, we configured the Acenic interfaces with MTUs of 1500 and 9000 bytes respectively. Otherwise, the test methodology remains the same as with our tests of interrupt coalescing. See [10] for a complete discussion of the implementation, test methodology and results. Figure 2 shows that jumbo frames do relieve both interrupt inter-arrival and stack processing pressures and thereby increase the effective utilization of the host processor.

Using jumbo frames increases effective utilization in high-performance computing nodes. Unfortunately, this method requires special routers and special TCP/IP stacks. Even though most Gigabit switches support jumbo frames, the different frame size decreases interoperability and increases costs. Finally, frames of only 9000 bytes will not contribute significantly to solving comparable problems created by a 10 Gb/s network, where inter-arrival times will only increase from 1.2μs to 7.2μs and the bottleneck will return. Jumbo frames are inadequate primarily because they are not fully interoperable

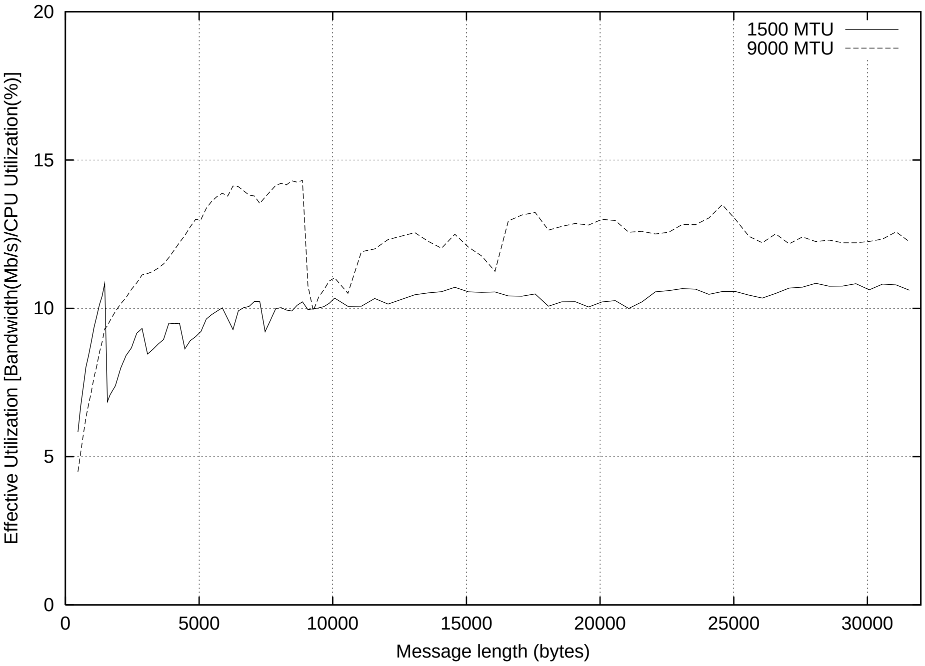

Figure 2. Effective utilization for 1500 byte Ethernet frames and 9000 byte Ethernet frames

(and therefore costly) and they are not scalable. These results are summarized in Table 4.

Performance Metrics	Jumbo Frames
increased effective utilization	fair
application-level flexibility	good
interoperability	poor
cost	poor
perf. scalability	*poor*
sys. scalability	fair

Table 4. Performance parameters for Jumbo Frames

4.3 Offload Communication Work onto the Network Interface Card

More and more work is being offloaded onto NICs, even in the commodity world of TCP/IP over Ethernet. One of the most successful examples of offloading communication processing is crypto hardware accelerators which have made virtual private networks (VPN) feasible. While a great deal of the work is offloaded directly into hardware (e.g. FPGA's), cards like the Myrinet NICs provide extensive computing resources and a rich development environment. Offloading work onto the NIC is a powerful method to decrease CPU overhead on compute nodes, but there is always the pressures of available resources on the NIC versus price per NIC.

Certainly, MAC assist, cyclic redundancy checks, and checksumming are the most common parts of the the TCP/IP protocol stack to be moved to the NIC, but many other elements of TCP/IP are being migrated off the host processor. For example, interrupt coalescing has been successfully deployed in hardware and recently researchers and companies have been offloading the entire TCP/IP protocol stack onto hardware[33, 32]. We see the trade-offs from offloading some of the IP stack onto the NIC versus offloading all of it onto the NIC.

4.3.1 Offloading IP Fragmentation and Reassembly

Fragmentation has been virtually nonexistent since Kent and Mogul [14] identified serious problems with intermediate fragmentation (where routers fragment packets too large for the outgoing MTU of the next link) and proposed path MTU discovery in order to avoid any fragmentation by intermediate routers between two endpoints. Endpoint fragmentation is the only kind of fragmentation allowed in IP version 6.

Endpoint fragmentation does not share most of the disadvantages of intermediate fragmentation. Endpoint fragmentation is already commonly used by applications such as NFS that benefit from large datagrams being sent and received at higher levels of the protocol stack (in the case of NFS because disk blocks are the smallest unit of currency for disks). Moreover, fragmentation and reassembly are tasks that are extremely well-suited for implementation on a reasonably powerful NIC such as the Acenic[1]. They are easy to separate from the rest of the IP stack. This is accomplished by transparently accepting packets larger than the real MTU of the link and fragmenting on the card. Almost no modification of the driver is necessary.

To test the unmodified firmware, we configured the interface with an MTU of 1500 bytes. In order to test our firmware, we configured the interface with an MTU of 64000 bytes, and configured the underlying firmware to use an actual MTU of 1500 bytes. In all other respects, our tests remain the same. See [10] for a complete discussion of the implementation, test methodology and results.

Figure 3 shows that offloading fragmentation and reassembly onto the NIC significantly increases the effective utilization of the host. Effective utilization increased because this method maintained bandwidths within about 10% of bandwidths using a 1500 byte MTU but reduced the amount of time the host spent processing communication by about 50%. In fact, offloading fragmentation and reassembly was more successful in relieving the interrupt pressure bottleneck than the use of interrupt coalescing or jumbo frames.

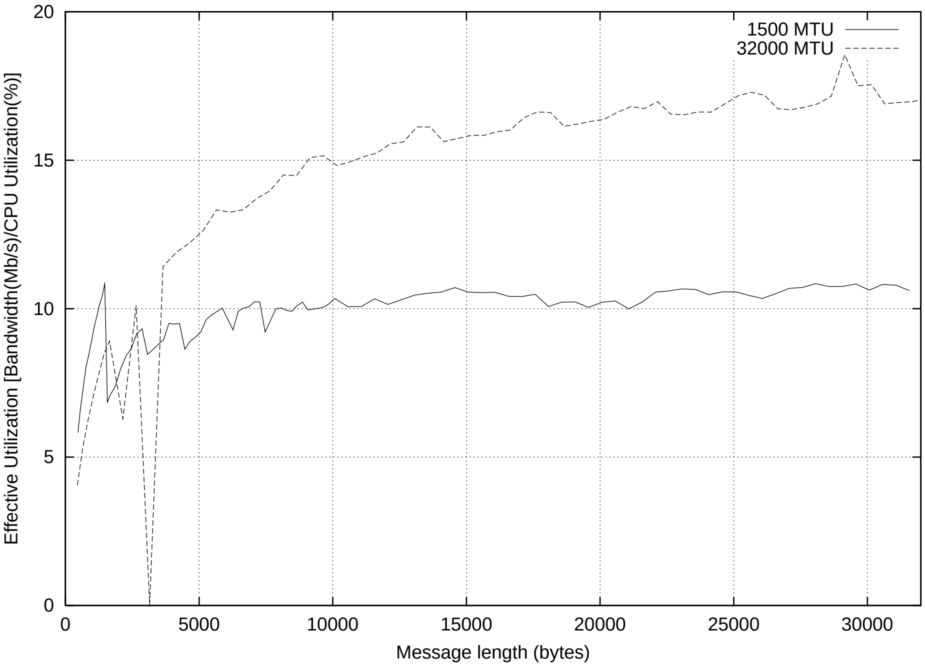

Figure 3. Effective utilization for unmodified firmware at 1500 byte MTU and offloaded fragmentation with driver MTU of 32000 bytes

We were able to implement fragmentation and reassembly on the NIC to demonstrate that it is possible to drastically impact the performance of a commodity protocol without significantly compromising its advantages[10]. Offloading a small part of the protocol led to a successful relieving of interrupt pressure while maintaining bandwidth, flexibility, interoperability and cost. Table 5 summarizes these results.

Performance Metrics	Offloaded Fragmentation
increased effective utilization	good
application-level flexibility	good
interoperability	good
cost	good
perf. scalability	good
sys. scalability	fair

Table 5. Performance parameters for Offloaded Fragmentation and Reassembly

4.3.2 Offloading TCP/IP

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[33, 32]. However, there are serious limitations in 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. Finally, 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.

Table 6 shows that while we definitely get a performance advantage[7], we lose the advantages of commodity protocols.

Performance Metrics	Offloaded TCP/IP
increased effective utilization	good
application-level flexibility	poor
interoperability	fair
cost	poor
perf. scalability	fair
sys. scalability	*poor*

Table 6. Performance parameters for Offloaded TCP/IP

4.3.3 Why Offload to the NIC?

Offloading large portions of the communication stack onto the NIC can be prohibitive due to limitations of the resources of the NIC itself. Researchers have two choices, decrease the amount of work to be offloaded or increase the resources of the NIC. We will continue to discuss how to decrease the amount of work to be offloaded, but additionally it is worth discussing the possibility of moving to a commodity co-processor rather than an intelligent NIC.

Communication co-processors are cheaper and more easily available than intelligent NICs. They are easy to maintain and the development environment is simpler. Communication co-processors allow for potential benefits in effective utilization without losing in cost or performance scalability.

One of the largest problems with using communication co-processors is that often application developers use the communication co-processor as a compute node in order to increase the scale of the system. Additionally, these applications often do not take advantage of more effective communication between dual processors. Changes to the communication stack and driver would need to be extensive in order to take full advantage of the dual processor design and specifically, the process scheduler would need to be modified.

While none of the above concerns are potentially prohibitive, from a research standpoint, using the NIC provides some real advantages. Offloading work onto a co-processor would not limit the amount of resources that could be used. Limiting the resources that can be used for communication allows us to get the most advantages from the least resources which is really the purpose of splintering.

What we'd really like to be able to do is get a grasp on what is the most optimum level of resources needed outside of the compute node in order to gain as much advantage from offloading as possible. Additionally, working with limited resources allows us to make real determinations about what functionality can be splintered. Finally, we will have a better understanding on the relative advantages of each splinter of functionality.

The impact of the research with respect to communication co-processors or intelligent NICs is twofold. First, there will be a better understanding of the amount of resources necessary to effectively offload work and second, there will be an understanding of the order in which functionality is best offloaded.

4.4 Summary

Using the interrupt coalescing method for relieving interrupt pressure required that small messages wait to be processed. Because offloaded fragmentation and reassembly assumes an MTU of *up to* 32000 bytes, small messages are unaffected by the method. Offloading leads to a larger performance gain than interrupt coalescing and also maintains application-level flexibility by not interfering with small messages.

The jumbo frames method for relieving interrupt pressure required special IP routers that allowed 9000 byte frames. Because offloaded fragmentation and reassembly assumes a wire MTU of 1500 bytes, no special routers are needed. This not only makes this method more interoperable, but it also makes the infrastructure required for this method much cheaper.

Finally, while offloading the TCP/IP stack provides the same performance advantages of offloading fragmentation and reassembly, the cost in loss of flexibility and resource usage is high. Specifically, there is no way to scale offloaded TCP/IP to large clusters since the memory required to manage thousands of connections cannot be available at the NIC.

Offloading IP fragmentation and reassembly increases the effective utilization of a host processor by reducing interrupts while maintaining high bandwidth. Furthermore, this method allows for application flexibility by not imposing latency restriction and remains fully interoperable with low resource usage. As we see in Table 7, offloading a portion of the IP protocol proves to be the best current solution.

Performance Metrics	Interrupt Coal	Jumbo Frames	Offload Frag	Offload TCP
utilization	fair	fair	good	good
flexibility	poor	good	good	poor
interoperability	good	poor	good	poor
low cost	good	poor	good	poor
perf.scalability	fair	poor	good	good
system scale	fair	fair	fair	fair

Table 7. Performance Metrics for All Solutions

5 Proposed Research

We have successfully splintered and offloaded the part of the IP stack in order to reduce the interrupt pressures associated with high-performance computing and to increase the effective utilization of compute nodes. Next, we will propose to splinter and offload part of the TCP stack in order to reduce the connection-setup costs associated with commodity protocols and increase the scalability of high-performance networks.

5.1 Reducing Connection-Setup Costs

Despite the impressive increases in performance of commodity TCP/IP in cluster computing using zero-copy mechanisms and techniques for decreasing interrupts, one of the fundamental drawbacks to TCP is the connection-oriented nature of the protocol. In large clusters with thousands of nodes, the expense of setting up, maintaining and tearing down connections can add a great deal to the communication overhead of the cluster. On the other hand, TCP has well-tested congestion and flow control mechanisms that provides fairness and reliability. Can we provide these advantages to environments in which thousands of nodes need to communicate with each other?

We propose splintering the functionality of the TCP congestion avoidance and flow control mechanisms (hereafter referred to as the congestion control mechanism) for three reasons. First, by using the most sophisticated techniques in TCP-friendly congestion control research, we will be able to decrease the amount of state necessary to maintain a connection and increase the speed of connection startup. Second, by offloading this congestion control onto the NIC, we will be able to decrease the CPU overhead of the host and decrease the latency of small messages. Third, by splintering the congestion control from the TCP stack we can provide hooks so that other protocols (like MPI) may use the functionality of congestion control without the unwanted overhead (like a memory copy) of TCP.

5.2 TCP Congestion and Flow Control

Before we can propose optimizations to the TCP stack, we review its basic congestion control mechanism.

The TCP protocol consists of two phases, the slow start phase and the congestion avoidance phase. The implementation of these two algorithms requires the maintenance of both a congestion window (*cwnd*) and a slow start threshold (*ssthresh*). The *cwnd* maintains the current number of bytes that have been sent but not acknowledged as received. The *ssthresh* is set to half of the size of number of bytes in flight when congestion occurs.[28]

When slow start occurs, the sender begins with a *cwnd* starting at one segment. With each acknowledgment of data received, the sender doubles the *cwnd* so the increase in the amount of data sent is exponential. This process continues until there is an indication of congestion. Once there is an indication of congestion (by duplicate acknowledgment of the same data), slow start is completed and congestion avoidance begins. In congestion avoidance, the *cwnd* is increased by 1/*cwnd* with each received acknowledgment so the increase in the amount of data sent is additive. If more than two duplicate acknowledgments are received, the *cwnd* is decreased to slightly more than the new *ssthresh*.

The difficulties with this congestion control mechanism are two-fold. When a connection is first established, there is traditionally no information and so slow start must be performed. This is an expensive process. In order to avoid the expense of ramping up from a beginning window of one segment, one can keep a connection open and maintain the existing *cwnd* and *ssthresh* state for each connection. However, in a cluster with thousands of potential connections, the amount of resources necessary to maintain so many connections can be prohibitive.

5.3 Shared Congestion Information

We propose to use methods that share congestion information to take advantage of the characteristics of high-performance networking and increase performance.

We want to decrease the amount of bandwidth wasted by starting a new connection and we want to decrease the amount of memory necessary to maintain an old connection. With the techniques of some of the modern research, we can decrease both the start-up cost and the cost of maintaining a connection by having connections share a *cwnd* and an *ssthresh*.

In wide-area networks, sharing connection information is problematic since congestion information may vary widely. However, with local-area networks, this variance is much less pronounced and congestion control is more homogeneous both across space (different hosts) and across time (to the same host). By sharing information, high-performance TCP can create and destroy connections with much less expense (because there is no slow start) and therefore maintain communication with thousands of nodes.

In a large cluster, the proposal that congestion information will be valid across thousands of connections is naive. With a single window for all connections, a failure at one host could result in a catastrophic slowdown as the congestion windows at all connected hosts stall waiting for the appropriate timeouts. We propose to study several ways to filter congestion information so that we need not maintain state for each connection, but do not suffer major inefficiencies.

While the final goal of splintering congestion control is to increase the scalability metric for TCP, sharing congestion information will more immediately increase the effective utilization of a host during the start of a connection. If we can significantly increase the effective utilization of the host during short message transfers (which will accentuate the connection-startup bandwidth) we will have succeeded in demonstrating the usefulness of shared congestion information.

5.3.1 Implementation

The Linux 2.4.17 kernel's TCP implementation maintains over 400 bytes of state per connection. For a cluster of tens of thousands of nodes where no state is shared, maintain all necessary connections will require over 4 Mb of memory.

There are two ways that a high-performance application can establish connections. They can serially open connections to all nodes in the communication block at the start of the application or they can open a connection when it is needed (and then either keep it open or close it). The first solution is an

n^2 solution since all nodes in the communication block must open connections to each of the others. If all of the connections are being opened at the same time, there is the additional cost of collisions in the network. Startup time for an application using this solution is very high.

On the other hand, an application library can open a connection only when it is needed which eliminates the long startup time. This solution requires slow start in the middle of the application run which adds unexpected latency and wastes bandwidth. If we can reduce the cost (in variance of latency and loss of bandwidth) of slow start, we can gain the advantages of opening connections as needed without paying the price.

I propose to share the timing information (over 75 bytes) and the congestion information (over 40 bytes) of connections in order to avoid slow start. We can share this information across nodes with similar distances. In the tree structure of clusters, this means that congestion and timing information can be shared among offspring of the same parent in a tree and potentially among all nodes in the same level of the tree. I will test both of these possibilities.

I propose to statically create congestion and timing structures for equivalence classes based on the parent nodes of the routing structure of the cluster. I will ignore timing irregularities and errors. I will test startup costs and effective utilization versus starting all connections at the beginning and starting connections as needed with no shared information. I propose next to statically create congestion and timing structures for equivalence classes based on levels of the routing structure of the cluster and will test this solution as well.

Additionally, as time affords, I will investigate ways to make this sharing of information dynamic (based on RTT variance), architecture independent, and fault tolerant.

5.4 Offloaded Congestion Control

After we have tailored the congestion control mechanism to the needs of high-performance computing, we propose offloading the mechanism to a NIC. The processing associated with maintaining the sliding window will no longer be at the expense of compute-bound applications on the host.

In addition to offloading the processing involved with maintaining congestion control information, offloaded congestion control facilitates true zero-copy TCP. Currently, the TCP stack can easily be modified to send messages without any data copy (the checksum is calculated during DMA to the NIC). However, zero-copy receives still require some form of copy or memory management (like pre-posted buffers or fbufs). Implementations of zero-copy TCP receive split the headers and the data at the NIC and send the headers through the operating system while moving the data directly to user space. This is a good solution, but still requires a context switch into kernel space for normal congestion control procedures. With congestion control on the NIC, the solutions which separate headers from data will now avoid the memory copy *and* the interrupt into the kernel.

Successful offloading of congestion control will again increase effective utilization over both normal TCP/IP messaging and over TCP/IP messaging with shared congestion control information. While shared congestion control information gained bandwidth without reducing CPU overhead, offloading the mechanism will reduce interrupts to the kernel which further reduces CPU overhead. Effective overhead will be increased. Currently we do not plan to implement a full zero-copy TCP receive, although we will design our offloaded TCP congestion control mechanism with a future zero-copy stack in mind.

5.5 Flexible Congestion Control

Finally, because often protocols and applications want the congestion control mechanism of TCP but wish to provide separate recovery mechanisms or don't want the overhead associated with sockets, we want to provide hooks directly into the congestion control mechanism. Ideally, the offloaded mechanism will provide congestion control for all applications on the host regardless of their communication protocol.

If we can accomplish providing hooks into the congestion control mechanism, we can show that MPI, for example, can use the management resources of the congestion control mechanism directly and concurrently. Success of this part of the research will be measured mostly at the design and specification stage of the research. The ability to provide an efficient and useful interface to the congestion control mechanism will be illustrated by the successful implementation of a small subset of MPI calls.

6 Related Work

We have placed each section of our initial results in a larger context of research. Work related to our proposal falls into three categories: connection information stored per connection, connection information shared across connections and hooks to allow application control of connections. This work is motivated by performance inefficiencies in the WAN world. We will elaborate on the motivation and discuss each category individually.

Research into speeding up the slow start mechanism in TCP has been very active in the last several years due primarily to the use of TCP/IP in the WAN world. Internet traffic tends to be small and bursty and often connections are closed before slow start completes meaning that Internet hosts are not exploiting their full bandwidth. Researchers in this area are concerned with utilizing more bandwidth for short messages. They are not concerned, as we are, with the amount of state that must be maintained or the amount of processing overhead added to accomplish this.

6.1 Connection Information Cached per Connection

One of the proposed solutions to the problem of slow start is to cache some information about a connection for use if another connection between the hosts is opened at a later time. Transaction TCP[6] caches the TCP control block (TCB) state which is, roughly, the MSS (maximum segment size), the smoothed RTT (round-trip time), the RTT variance and sequence number for a given connection so that upon opening the connection again, the three-way handshake is no longer needed.[28] In this case, the *ssthresh* and *cwnd* are not cached.

Several variations on the theme of caching TCB state and congestion information for each connection have been introduced[34]. TCP Fast Start[20] caches the *cwnd* and *ssthresh* along with TCP control block state, like [31], so it is able to perform fast start when cached information is valid, but it also can move into slow start with little extra overhead if the network is experiencing additional congestion. These caching methods are backward-compatible solutions which decrease connection setup time. However, the problem of having to maintain too much state is exacerbated rather than relieved, and these methods alone will fail to relieve the problems associated with TCP in high-performance computing.

6.2 Connection Information Shared across Connections

Two methods which do decrease the amount of necessary state while still decreasing setup time are simply increasing the initial congestion window and caching information spatially instead of temporally. [2] proposes increasing the congestion window from one segment to the minimum of four segments or 4Kb. If we consider using the current IP stack, the MSS is 1500 bytes and so the congestion window would start at 4Kb. If we consider using the offloaded IP fragmentation and reassembly, we can alter the MSS on the fly and the MSS can be adapted to become more closely related to the congestion window. While the 4Kb initial congestion window is static and conservative, the spirit of [2] found in the proposal to adapt the size of the segment sent (and continuing to send only one) will be tested.

The TCP-Int[3] method caches the *ssthresh* and *cwnd* for a group of connections. Many Internet application open multiple connections between the same hosts and the congestion information can be shared between these connections. TCP-SPAND[34] suggests sharing congestion information across hosts using passive observation of traffic. These are similar to our proposal although we plan to share congestion information between connections to different receivers within the same sender. We may chose to explore sniffing TCP traffic like TCP/SPAND, but we will not employ CPU resources to do so. Finally, both TCP-Int and TCP/SPAND are integrated into the TCP stack on the kernel while we plan on moving our state maintenance off of the host and onto a NIC.

6.3 Providing Control to the Application

The Congestion Manager (CM)[4] is a well-developed system that allows for applications to have hooks into the congestion management scheme. These hooks allow for applications to have an influence in how bandwidth is divided between connections. Additionally, all traffic flows through the CM which multiplexes and demultiplexes flows. CM provides hooks by means of call-back mechanisms and socket-setup variables and the algorithms are independent of any transport protocol. CM is not globally compatible with TCP as its congestion avoidance algorithm is aggressive whereas our plans are to use aggressive techniques only within the cluster (or LAN). Also, CM relies on additionally feedback from the receiver which makes it less desirable in an environment control communication must be kept to a minimum. Finally, CM adds a great deal of additional communication overhead rather than decreasing it as our offloaded congestion control mechanism will do.

7 Impact

There has long been a trend toward improving the performance of TCP/IP in high-performance computing while still maintaining its tremendous advantages. By tailoring recent advances in TCP in the WAN world to the high-performance arena, we can show that we can reduce overhead and maintenance of state enough that TCP/IP becomes a viable protocol in the world of cluster computing. Furthermore, by appropriately offloading small parts of the protocols' functionality onto a NIC we have demonstrated the methods of splintering that will become more and more prevalent as we move into a distributed computing environment. Finally, by providing hooks into the congestion control mechanism we have shown that while offloading will be the key to increased efficiency, splintering is the key to bridging the gap between commodity and high-performance protocols.

Here we have found that splintering the protocol stacks and offloading some implementation aspects of IP will lower communication overhead and increase performance without sacrificing the interoperability and cost advantages of the protocol. Finally, we propose a method to decrease connection management resources in TCP and thereby increase scalability without sacrificing the advantages that commodity protocols offer.

References

[1] Acenic gigabit ethernet for linux. Web: 'http://jes.home.cern.ch/jes/gige/acenic.html', August 2001.

[2] M. Allman, S. Floyd, and C. Partridge. RFC 2414 : Increasing TCP's initial window, September 1998. Status: EXPERIMENTAL.

[3] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. Katz. TCP behavior of a busy internet server: Analysis and improvements. In *IEEE INFOCOM*, March 1998.

[4] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for internet hosts. In *SIGCOMM*, pages 175–187, 1999.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[6] R. Braden. RFC 1379: Extending TCP for transactions - concepts, November 1992. Status: EXPERIMENTAL.

[7] 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.

[8] H. K. J. Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[9] S. Elbert, Q. Snell, A. Mikler, G. Helmer, C. Csandy, K. Stearns, B. MacLeod, M. Johnson, B. Osborn, and I. Verigin. Gigabit ethernet and low cost supercomputing. Technical Report 5126, Ames Laboratory and Packet Engines, Inc., 1997.

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

[11] P. Hochmuth. Vendors lower gigabit ethernet price bar. Web: 'http://www.nwfusion.com/archive/2001/127651_11-26-2001.html', 2001.

[12] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP extensions for high performance, May 1992. Obsoletes RFC1072, RFC1185 [13]. Status: PROPOSED STANDARD.

[13] V. Jacobson, R. T. Braden, and L. Zhang. RFC 1185: TCP extension for high-speed paths, Oct. 1990. Obsoleted by RFC1323 [12]. Status: EXPERIMENTAL.

[14] C. A. Kent and J. C. Mogul. Fragmentation considered harmful. *WRL Technical Report 87/3*, Dec. 1987.

[15] C. Kurmann, M. Muller, F. Rauch, and T. Stricker. Speculative defragmentation — a technique to improve the communication software efficiency for gigabit ethernet. In *Proc. 9th IEEE Symp. High Performance Distr. Comp.*, pages 131–138, August 2000.

[16] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Symposium on Operating Systems Principles*, pages 244–255, 1993.

[17] 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.

[18] MPI Forum. Mpi:a message-passing interface. In *International Journal of Supercomputer Applications*, 1994.

[19] The public netperf homepage. Web: 'http://www.netperf.org/netperf/NetperfPage.html', 2001.

[20] V. Padmanabhan and R. Katz. TCP Fast Start: a technique for speeding up web transfers. In *IEEE Globecom '98 Internet MiniConference*, November 1998.

[21] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, 1996.

[22] C. Papadopoulos and G. M. Parulkar. Experimental evaluation of SUNOS IPC and TCP/IP protocol implementation. *IEEE/ACM Transactions on Networking*, 1(2):199–216, 1993.

[23] V. Paxon. End-to-end internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7, 3, 1999.

[24] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local-area communication with fast sockets. In *Usenix Annual Technical Conference*, pages 257–274, 1997.

[25] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. Maccabe, L. Fisk, and T. Stallcup. The puma operating system for massively parallel computers. In *1995 Intel Supercomputer User's Group Conference*, 1995.

[26] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, 1993.

[27] P. Steenkiste. A systematic approach to host interface design for high-speed networks. *IEEE Computer*, 27(3):47–57, 1994.

[28] W. R. Stevens. *TCP/IP Illustrated, Volume 1; The Protocols*. Addison Wesley, Reading, 1994.

[29] W. Toomey. *A Rate-Based Congestion Control Framework for connectionless Packet-Switched Networks*. PhD thesis, University of New South Whales, 1997.

[30] L. Torvalds. Re: zero-copy tcp fileserving. Web: 'http://www.tux.org/hypermail/linux-kernel/1999week23/1085.html', 1999. Email from Linus Torvalds to Zach Brown from the Linux Kernel Mailing List.
[31] J. Touch. RFC 2140: TCP control block interdependence, April 1997. Status: EXPERIMENTAL.
[32] Wind river networking technologies: Tornado for intelligent network acceleration. Web: 'http://www.windriver.com/products/html/tina.html', 2002.
[33] Alacritech inc. Web: 'http://www.alacritech.com', 2002.
[34] Y. Zhang, L. Qiu, and S. Keshav. Optimizing TCP start-up performance. Technical Report TR99-1731, 10, 1999.

Timeline

Month	Tasks	Milestones
February March	Create test environment	 Run control test on Linux
April	Implement TCP-Int and large MSS	
May June	Design and test various filters for sharing congestion information	Test TCP-Int and large MSS
July	Design offloaded congestion control mechanism	
August September	Offload congestion control mechanism	 Test offloaded congestion control
October November	 Write dissertation	
December		
January February March April		 Defend dissertation