Optimizing TCP Receive Performance

Aravind Menon and Willy Zwaenepoel
School of Computer and Communication Sciences
EPFL

Abstract

The performance of receive side TCP processing has traditionally been dominated by the cost of the 'per-byte' operations, such as data copying and checksumming. We show that architectural trends in modern processors, in particular aggressive prefetching, have resulted in a fundamental shift in the relative overheads of per-byte and per-packet operations in TCP receive processing, making per-packet operations the dominant source of overhead.

Motivated by this architectural trend, we present two optimizations, receive aggregation and acknowledgment offload, that improve the receive side TCP performance by reducing the number of packets that need to be processed by the TCP/IP stack. Our optimizations are similar in spirit to the use of TCP Segment Offload (TSO) for improving transmit side performance, but without need for hardware support. With these optimizations, we demonstrate performance improvements of 45-67% for receive processing in native Linux, and of 86% for receive processing in a Linux guest operating system running on Xen.

1 Introduction

There is a large body of research on techniques to improve the performance of the TCP stack. Although a number of techniques have been proposed for improving the performance of the transmit side in TCP, such as zero-copy transmit and segmentation offload, there has been relatively little work on improving the receive side performance. New applications, such as storage area networks, make receive performance a possible concern. In this paper we analyze the receive side performance of TCP on modern machine architectures, and we present new mechanisms to improve it.

Conventionally, the dominant source of overhead in TCP receive processing has been the cost of the 'per-byte' operations, those operations that touch all bytes of input data, such as data copying and checksumming [3, 9, 6, 12]. In contrast, the 'per-packet' operations, that are proportional to the number of packets processed, such as header processing and buffer management, were shown to be relatively cheap. Thus, unlike in TCP transmit processing, in TCP receive processing, there has been little emphasis on reducing the per-packet overheads.

The high cost of the per-byte operations, at least in older architectures, resulted in part from the fact that the network interface card (NIC) places newly arrived packets in main memory. All operations that touch the data of the packet thus incur compulsory cache misses. Modern processors, however, use aggressive prefetching to reduce the cost of sequential main memory access. Prefetching has a significant impact on the relative overheads of the per-byte and per-packet operations of TCP receive processing. Since the per-byte operations, such as data copying and checksumming, access the packet data in a sequential manner, their cost is much reduced by prefetching. In contrast, the per-packet operations, which access main memory in a non-sequential, random access pattern, do not benefit much. Thus, as the per-byte operations become cheaper, the per-packet overheads of receive side TCP processing become the dominant performance component.

Motivated by these architectural trends in modern processors, we present two optimizations to the TCP receive stack that focus on reducing the per-packet overheads. Our optimizations are similar in spirit to the TCP Segment Offload (TSO) optimization used for improving TCP transmit side performance. Unlike TSO, however, which requires support from the NIC, our optimizations can be implemented completely in software, and are therefore independent of the NIC hardware.

The first optimization is to perform packet 'aggregation': Multiple incoming network packets for the same TCP connection are aggregated into a single large packet, before being processed by the TCP stack. The cost of

packet aggregation is much lower than the gain achieved as a result of the TCP stack having to process fewer packets. TCP header processing is still done on a per-packet basis, because it is necessary for aggregation, but this is only a small part of the per-packet overhead. The more expensive components, in particular the buffer management, are executed once per aggregated packet rather than once per network packet, hence leading to a considerable overall reduction in cost.

In addition to aggregating received packets, we present a second optimization, TCP acknowledgment offload: Instead of generating a sequence of acknowledgment packets, the TCP stack instead generates a single TCP acknowledgement packet template, which is then turned into multiple TCP acknowledgment packets below the TCP stack. The gain in performance comes from the reduction in the number of acknowledgment packets to the processed by the transmit side of the TCP stack.

While the two optimizations are logically independent, receive aggregation creates the necessary condition for TCP acknowledgement offload to be effective, namely the need to send in short succession a substantial number of near-identical TCP acknowledgment packets.

We have implemented these optimizations for two different systems, a native Linux operating system, and a Linux guest operating system running on the Xen virtual machine monitor. We demonstrate significant performance gains for data-intensive TCP receive workloads. We achieve performance improvement of 45-67% in native Linux, and 86% in a Linux guest operating system running on Xen. Our optimizations also scale well with the number of concurrent receive connections, performing at least 40% better than the baseline system. Our optimizations have no significant impact on latency-critical workloads, or on workloads with small receive message sizes.

The organization of the rest of the paper is as follows. In section 2, we analyze the receive side processing overhead of TCP in native Linux and in a Linux guest operating system running on Xen, and show the significance of per-packet overheads in both settings. We present our optimizations to the TCP receive stack in sections 3 and 4 We evaluate our optimized TCP stack in section 5, and discuss related work in section 6. We conclude in section 7.

2 Background

We start by studying the impact of aggressive prefetching used by modern processors on the relative overheads of per-packet and per-byte operations in TCP receive processing. We profile the execution of the TCP receive stack in three different systems, a Linux uniprocessor system, a Linux SMP system, and a Linux guest operating system running on the Xen virtual machine monitor (VMM). We show that in all three systems the per-packet overheads have become the dominant performance component.

We next analyze the per-packet overheads on the receive path in more detail. We show that TCP/IP header processing is only a small fraction of the overall per-packet overhead. The bulk of the per-packet overhead stems from other operations such as buffer management. This observation is significant because TCP/IP header processing is the only component of the per-packet overhead that must, by necessity, be executed for each incoming network packet. The other operations are currently implemented on a per-packet basis, but need not be. The bulk of the per-packet overhead can therefore be eliminated by aggregating network packets before they incur the expensive operations in the TCP stack.

In the experiments in this section, we focus on analyzing bulk data receive workloads, in which all network packets received are of MTU size (1500 bytes for Ethernet). We use a simple netperf [1] like microbenchmark, which receives data continuously over a single TCP connection at Gigabit rate. Profiling provides us with the breakdown of the execution time across the different subsystems and routines.

The experiments are run on a 3.80 GHz Intel Xeon dual-core machine, with an Intel e1000 Gigabit NIC. The native Linux kernel version used is Linux 2.6.16.34, and for the virtual machine experiments we use Linux 2.6.16.38 running on Xen-3.0.4. Profile statistics are collected and reported using the OProfile [2] tool.

2.1 Impact of Prefetching

To counter the growing gap between processor and main memory speeds, modern processors use aggressive prefetching. The impact of this architectural trend on TCP receive processing is that per-packet overheads become dominant, while the per-byte overheads become less important. This can be seen from figure 1, which shows the breakdown of TCP receive processing overhead in a native Linux system, as a function of the extent of prefetching enabled in the CPU.

The total processing overhead is divided into three categories: the per-byte data copying routines, `per-byte`, the per-packet routines, `per-packet`, and other miscellaneous routines, `misc`. The `misc` routines are those that are not really related to receive processing, or cannot be classified as either per-packet or per-byte, for example scheduling routines. The per-packet routines include the device driver routines along with the TCP stack operations.

The three groups of histograms in figure 1 show the breakdown of the receive processing overhead for dif-

ferent CPU configurations: The `None` configuration uses no prefetching, `Partial` uses adjacent cache-line prefetching, and `Full` uses adjacent cache-line prefetching and stride-based prefetching.

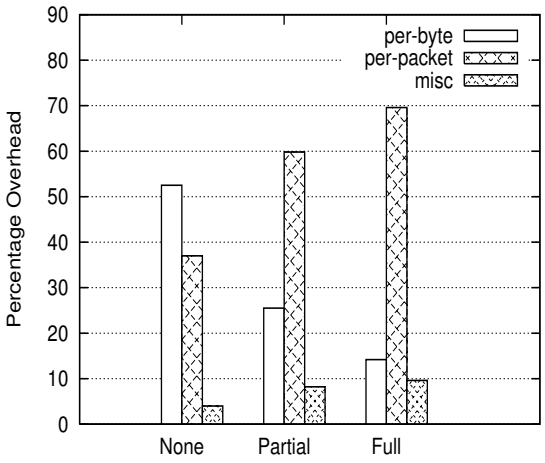

Figure 1: Impact of Prefetching on the Relative Cost of Per-Byte and Per-Packet Operations in TCP Receive Processing on a Uniprocessor as a Function of the Degree of Prefetching

As the CPU is configured to prefetch more aggressively, the contribution of the per-byte operations to the overall overhead declines from 52% to 14%. The proportion of the per-packet operations in the overall overhead increases correspondingly from 37% to nearly 70%, and becomes more important than the per-byte overheads.

The increase in importance of the per-packet costs is a consequence of the architectural evolution of microprocessors to cope with the increasing gap between memory and CPU speeds. This evolution is present across different architectures and operating systems, and is likely to further increase with future processors.

Figure 2 shows the relative overhead of per-packet and per-byte operations for three different systems, all with aggressive prefetching enabled: a Linux uniprocessor system (`UP`), a Linux SMP system (`SMP`), and a Linux guest operating system running on the Xen VMM (`Xen`). As can be clearly seen, in all three systems, per-packet overheads far outweigh the per-byte overheads.

2.2 Per-Packet Overhead

We now look in more detail into the per-packet overheads of receive processing. The key objective here is to distinguish between the operations that are a necessary part of TCP receive processing and have to be done on a per-packet basis, such as TCP/IP header processing, and operations that are not central to receive processing

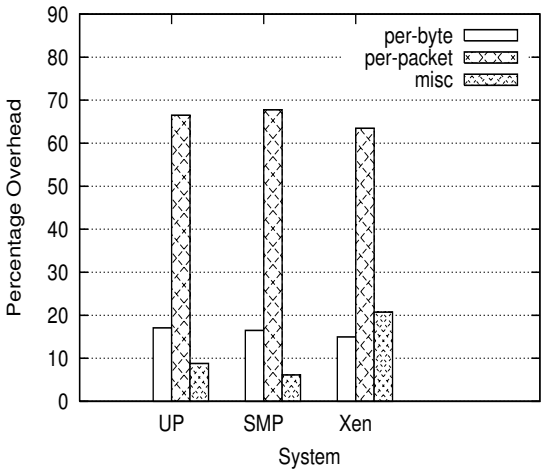

Figure 2: Per-byte vs. Per-packet Overhead in Uniprocessor, Multiprocessor and Virtualized Systems (with Full Prefetching Enabled)

or do not need to be executed on a per-packet basis, although for historical or architectural reasons they are implemented that way.

Figure 3 shows the breakdown of the TCP receive processing overhead into different kernel categories on Linux version 2.6.26.34, running on a uniprocessor 3 Ghz Xeon, with full prefetching enabled. The Y axis shows the number of busy CPU cycles per packet spent in the different routines. The X axis shows the different categories of overhead during receive processing.

The `per-byte` routines correspond to the per-byte operations in the receive path. The `misc` routines are those which are unrelated to receive processing, and are not strictly per-packet or per-byte. The per-packet overheads are split into five subgroups, `rx`, `tx`, `buffer`, `non-proto`, and `driver`, defined as follows:

1. **rx:** TCP/IP protocol processing routines on the receive path of the TCP stack.
2. **tx:** TCP/IP protocol processing routines on the transmit path of the TCP stack for transmission of ACKs.
3. **buffer:** Buffer management routines for network packets, ACK packets, and `sk_buffs`, the buffer metadata structure used in Linux.
4. **non-proto:** Other kernel routines which operate on per-packet basis, but are not part of the core TCP/IP protocol processing. Some of these are Linux-specific, such as routines for packet movement between `softirq` and `interrupt` context, whereas others are more generic, such as the packet filtering and network bridging routines.

5. **driver:** Device driver routines and routines running in interrupt mode.

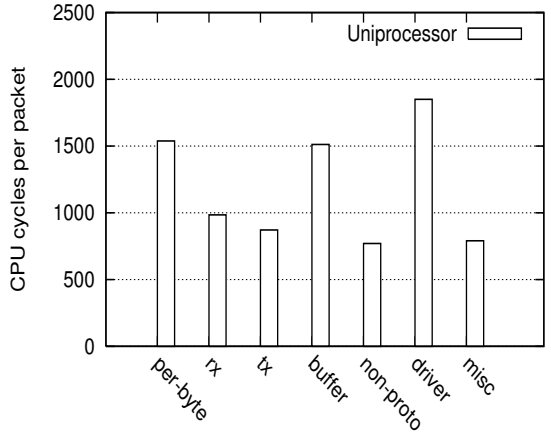

Figure 3: Breakdown of Receive Processing Overheads in a Uniprocessor system

The overhead of the device driver routines, `driver`, is roughly 21%. This overhead is per-packet, but that cannot be changed without modifications to the NIC. Thus, we henceforth distinguish between the `driver` routines and the other per-packet routines in the network stack, namely `rx`, `tx`, `buffer` and `non-proto`. Even excluding the driver, the overhead of these per-packet routines is 46%, which is much higher than the per-byte overhead of copying, 17%.

The overhead of the TCP/IP protocol processing itself, consisting of `rx` and `tx`, is only around 21% of the total overhead. The larger part of the per-packet overhead, around 25%, comes from the buffer management (`buffer`) and non-protocol processing related routines (`non-proto`) involved in handling of packets within the TCP/IP stack. Detailed profiling shows that most of the buffer management overhead is incurred in the memory management of `sk_buffs`, and not in the management of the network packet buffer itself. In conclusion, the bulk of the per-packet overhead in the network stack is incurred in routines which are not related to the protocol processing.

2.3 SMP Overheads

We now look in more detail at the per-packet overheads in an SMP environment. The system used is a Linux 2.6.16.34 SMP kernel running on dual-core Intel 3.0 GHz Xeon machine. Figure 4 shows the breakdown of the receive processing overheads in the SMP environment. Since the profile of the processing overhead on the SMP is very similar to that observed on the uniprocessor system, we present both profiles in the same figure to illustrate the impact of multiprocessing on the different components of the TCP stack.

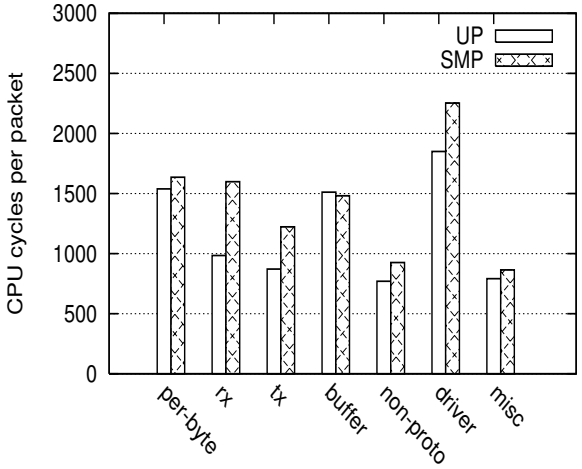

Figure 4: Breakdown of Receive Processing Overheads in an SMP vs. a UP environment

First, as for the uniprocessor system, the overhead of the per-packet routines (`rx`, `tx`, `buffer` and `non-proto`) is much higher (48%) than the per-byte copy overhead (16%). Second, in going from the uniprocessor to the SMP configuration, the per-byte copying overheads remain roughly the same, but there is a non-trivial increase in the overhead of some per-packet operations. In particular, in the SMP configuration the TCP receive routines (`rx`) incur 62%, and the TCP transmit routines (`tx`) incur 40% more overhead compared to the uniprocessor configuration. The buffer management routines, `buffer`, do not show significant difference in the two configurations.

The main reason for the increase in the overhead of the per-packet network stack routines is that in an SMP environment the TCP stack uses locking primitives to ensure safe concurrent execution. On the `x86` architecture, locking is implemented through the use of `lock`-prefixed atomic *read-modify-write* instructions, which are known to suffer from poor performance on the Intel `x86` CPUs.

In contrast, the per-byte data copy operations in the TCP stack can be implemented in a lock-free manner, and thus do not suffer from SMP scaling overheads. The buffer management routines do not suffer synchronization overheads because they are implemented in a mostly lock-free manner in Linux.

Thus, the locking and synchronization overheads of the TCP stack in an SMP environment are primarily in-

curred in the per-packet operations, and they grow in proportion to the number of packets that the TCP stack has to process. In conclusion, here again, mechanisms that reduce the number of packets handled by the TCP stack reduce these overheads.

2.4 Virtualization Overheads

We now analyze the receive processing overheads of the TCP stack in a virtual machine environment. In a virtual machine environment, device virtualization is an important part of the network I/O stack. We therefore analyze the extended network stack, including the virtualization stack.

Figure 5 shows a high level picture of the network virtualization architecture in Xen. Guest operating systems (guest *domains* in Xen) make use of a *virtual* network interface for network I/O, and do not access the *physical* network interface (NIC) directly. The *Driver* domain is a privileged domain that manages the physical network interface and multiplexes access to it among the guest domains. The *virtual* interface of the guest domain is connected to the physical NIC through a pair of *backend-frontend* paravirtualized drivers, and a network bridge in the driver domain. A more detailed description of the Xen networking architecture can be found in [7].

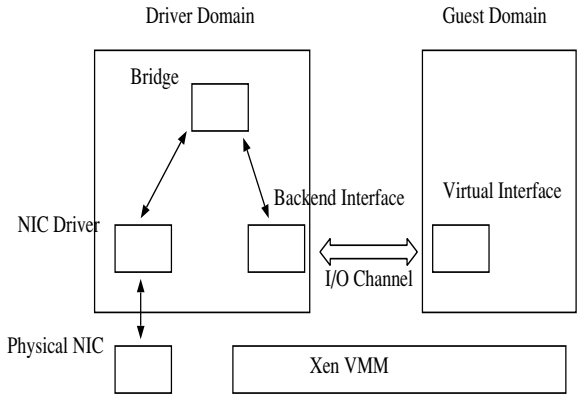

Figure 5: Xen I/O Architecture

Figure 6 shows the breakdown of the receive processing overhead for a Linux 2.6.16.38 guest domain running on Xen 3.0.4. The processing overheads in the guest domain, driver domain and Xen are split into the following categories:

1. **per-byte**: Data copy routines. This includes the two data copies on the receive path: the first from the driver domain into the guest domain, and second copy from the guest kernel into the guest application.
2. **non-proto**: This includes the bridge and netfilter routines in the driver domain, and similar non-protocol processing related routines in the guest domain. The bridge transfers packets received from the physical NIC to the backend interface. The overhead in this category is essentially a per-packet overhead.
3. **netback**: The netback driver initializes transfer of packets from the driver to the guest domain. Its overhead is mostly per-packet, and is proportional to the number of packet fragments it transfers.
4. **netfront**: The netfront driver receives packets from the driver domain and passes it onto the TCP stack. Its overhead is similar to the netback driver, and is proportional to the number of packet fragments it accepts.
5. **tcp rx and tx**: The transmit and receive TCP routines in the guest domain. These are per-packet overheads.
6. **buffer**: Buffer management routines, in both the driver domain and the guest domain. This is per-packet overhead.
7. **driver**: Device driver running in the Driver domain. This is a per-packet overhead.
8. **xen**: Xen manages domain scheduling, inter-domain interrupts, validation of packet transfer rights, etc. Its overhead cannot be classified strictly as either per-packet or per-byte.
9. **misc**: Other routines. These cannot be classified as either per-packet or per-byte.

The overall overhead of the per-packet routines in the receive path, comprising of the `non-proto`, `netback`, `netfront`, `tcp rx`, `tcp tx` and `buffer` routines, adds up to roughly 56% of the total overhead, and is significantly higher than the per-byte copy overhead, 14%. This is inspite of the fact that there are two data copies involved, one from the driver domain to the guest domain, and the second from the guest kernel to the guest application.

The major part of the per-packet overhead is incurred in the routines of the network virtualization stack, and only a small part of the per-packet overhead is incurred in the TCP protocol processing in the guest domain. Thus, the `non-proto` routines, the `netback` and `netfront` drivers, and the `buffer` management routines add up to 46% of the total overhead, whereas TCP/IP processing incurs only 10% of the total overhead.

Thus, here also, mechanisms that reduce the number of packets that need to be processed by the network stack

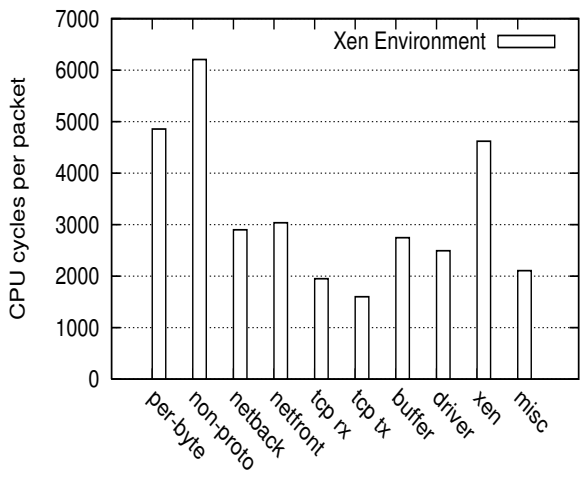

Figure 6: Breakdown of Receive Processing Overheads in Xen

can significantly reduce the overhead on the network virtualization path of guest domains.

2.5 Summary

First, architectural trends in microprocessors have resulted in a fundamental shift in the relative overheads of sequential and non-sequential memory access. Sequential memory access can be optimized by prefetching, while non-sequential access cannot. This has resulted in a fundamental shift in the relative overheads of per-byte and per-packet operations in TCP receive processing. While previously data copying was the primary performance bottleneck, in current systems the per-packet operations in the TCP receive path are the dominant overhead component. We have validated this trend for three different systems.

Second, reducing the number of packets to be processed by the TCP receive path holds promise as a solution for reducing the overhead of the per-packet operations. TCP/IP processing is the only overhead that needs to be incurred on a per-packet basis, but it is only a small component of the total overhead. The larger components of the overhead (system, virtualization, and SMP scaling) currently are incurred on a per-packet basis, but need not be.

We next present two optimizations to the TCP/IP stack that exploit these observations, Receive Aggregation and Acknowledgment Offload.

3 Receive Aggregation

The objective of Receive Aggregation is to reduce the number of packets that the network stack has to process on the receive path, while still ensuring that the TCP/IP protocol processing of the packets is done correctly.

The basic idea of Receive Aggregation is that, instead of allowing the network stack to process packets received from the NIC directly, the packets are preprocessed and coalesced into 'aggregated' TCP packets, which are then passed on to the network stack for further processing. Multiple 'network' TCP packets are aggregated into a single 'host' TCP packet, thereby reducing the number of packets that the network stack has to process.

Ideally, aggregation would be done entirely in a proxy between the driver and the TCP stack, and without any changes to the rest of the kernel code. For correctness and performance, this complete separation cannot be achieved, and small changes to the driver and the TCP layer are necessary. No changes were made to the IP layer or the layout of the kernel data structure for storing packets, `sk_buff` in Linux.

3.1 Which Packets are Aggregated

Receive Aggregation takes place at the entry point of the network stack. Packet coalescing is done for network TCP packets which arrive "in-sequence" on the same TCP connection. Thus, the incoming packets must have the same source IP, destination IP, source port, and destination port fields. The packets must also be in sequence, both by TCP sequence number and by TCP acknowledgment number. Thus, the sum of the TCP sequence number of one packet and its length must be equal to the TCP sequence number of the next packet. Also, a TCP packet later in the aggregated sequence must have a TCP acknowledgment number greater than or equal to that of a previous packet in the sequence.

Packet aggregation is done only for valid TCP packets, i.e., those with a valid TCP and IP checksum. We verify only the IP checksum field of the network TCP packet before it is used for aggregation. For the TCP checksum, we assume the common case that the NIC supports checksum offloading, and has validated the TCP checksum. This is because verifying the TCP checksum in software would make the aggregation expensive.

If the network card does not support receive checksum offloading, we do not perform Receive Aggregation.

TCP packets of zero length, such as pure ACK packets, are not aggregated. This simplifies the handling of duplicate ACKs in the TCP layer, and is discussed in section 3.6.

Since the TCP and IP headers support a large number of option fields, it is not possible to aggregate two

TCP packets if they contain different option fields. Also, the aggregation function becomes quite complicated if it has to support all possible TCP and IP options. Thus, for simplicity, we only aggregate TCP packets whose IP headers do not use any IP options or IP fragmentation, and whose TCP headers use only the TCP timestamp option.

Packets which fail to match any of the conditions for Receive Aggregation are passed unmodified to the network stack. In doing so, we ensure that there is no packet reordering between packets of the same TCP connection, i.e., any partially aggregated packet belonging to a TCP connection is delivered before any subsequent unaggregated packet is delivered.

3.2 Aggregated Packet Structure

Once a valid set of network packets is identified for aggregation, according to the conditions described above, the aggregation function coalesces them into an aggregated TCP packet for the network stack to process.

The aggregated TCP packet is created by 'chaining' together individual TCP packets to form the 'fragments' of the aggregated packet, and by rewriting the TCP/IP header of the aggregated packet. The TCP/IP header of the first TCP fragment in the chain retains its becomes the header of the aggregated packet, while the subsequent TCP fragments retain only their payload. The chaining is done in an OS specific manner. In Linux, for instance, chaining is done by setting the fragment pointers in the `sk_buff` structure to point to the payload of the TCP fragments. Thus, there is no data copy involved in packet aggregation.

The TCP/IP header of the aggregated packet is rewritten to reflect the packet coalescing. The IP packet length field is set to the length of the total TCP payload (comprising all fragments) plus the length of the header. The TCP sequence number field is set to the TCP sequence number of the first TCP fragment, and the TCP acknowledgment number field is set to the acknowledgment number of the last TCP fragment. The TCP advertised window size is set to the window size advertised in the last TCP fragment. A new IP checksum is calculated for the aggregated packet using its IP header and the new TCP pseudo-header. The TCP checksum is not recomputed (since this would be expensive), instead we indicate that the packet checksum has been verified by the NIC.

The TCP timestamp in the aggregated packet is copied from the timestamp in the last TCP fragment of the aggregated packet. Theoretically, this results in the loss of timestamp information, and may affect the precise estimation of RTT values. However, in practice, since only packets which arrive very close in time to each other are aggregated, the timestamp values on all the TCP fragments are expected to be the same, and there is no loss of precision. We give a more rigorous argument for this claim in section 3.6.

Finally, the aggregated TCP packet is augmented with information about its constituent TCP fragments. Specifically, the TCP acknowledgment number of each TCP fragment is saved in the packet metadata structure (`sk_buff`, in the case of Linux). This information is later used by the TCP layer for correct protocol processing.

3.3 When Aggregation Stops

Network Packets are aggregated as they are received from the NIC driver. The maximum number of network TCP packets that get coalesced into an aggregated TCP packet is called the Aggregation Limit. Once an aggregated packet reaches the Aggregation Limit, it is passed on to the network stack.

The actual number of network packets that get coalesced into an aggregated packet may be smaller than the Aggregation Limit, and depends on the network workload and the arrival rate of the packets. If an aggregated packet contains less than the Aggregation Limit number of network packets, and no more network packets are available for processing, then this semi-aggregated packet is passed on to the network stack without further delay. Thus, the network stack is never allowed to remain idle while there are packets to process in the system. Receive Aggregation is thus work-conserving and does not add to the delay of packet processing.

We expect the performance benefits of Receive Aggregation to be proportional to the number of network packets coalesced into an aggregated packet. However, the incremental performance benefits of aggregation are expected to be marginal beyond a certain number of packets. Thus, the Aggregation Limit serves as an upper bound on the maximum number of packets to aggregate, and should be set to a reasonable value at which most of the benefits of aggregation are achieved. We determine a good cut-off value for the Aggregation Limit experimentally.

3.4 Modifications to the TCP layer

The aggregated TCP packet is received by the network stack, and gets processed through the MAC and IP layers in the same way as a regular TCP packet is processed. However, at the TCP layer, modifications are required in order to handle aggregated packets correctly. This is because TCP protocol processing in the TCP layer is dependent on the actual number of 'network' TCP packets, and on the exact sequence of TCP acknowledgments received. Since Receive Aggregation modifies both these

values, changes are required to the TCP layer to do correct protocol processing for aggregated packets.

There are two specific situations for which the TCP processing needs to be modified:

1. Congestion Control: The congestion window of a TCP sender is updated based on the *number* of TCP Acknowledgment packets received by the sender, and not on the total number of *bytes* acknowledged by the receiver. Since Receive Aggregation sets the TCP ACK field in the aggregated packet to the ACK field of the last TCP fragment, the conventional TCP layer implementation would set the congestion window differently from what is expected in the absence of receive aggregation.

   The modified TCP layer computes the congestion window using the TCP acknowledgment numbers of all the TCP fragments of the aggregated TCP packet, instead of just using the final acknowledgment number. As noted in section 3.1, the acknowledgment numbers of the individual TCP fragments are stored in the packet metadata structure (`sk_buff`) when Receive Aggregation is performed.

2. TCP Acknowledgments: The TCP protocol specifies that an acknowledgment packet must be generated for every alternate full TCP segment received by the receiver. Since Receive Aggregation coalesces multiple network TCP packets into a single aggregated packet, the conventional TCP layer would concude that it has received only a single TCP segment, and therefore generate the wrong number of acknowledgment packets.

   The modified TCP layer computes the correct number of acknowledgments by taking into account the individual TCP fragments, instead of considering the whole aggregated packet as one segment. This information is also stored in the `sk_buff` structure during receive aggregation.

3.5 Implementation

Receive Aggregation is implemented at the entry point of the network stack processing routines. For the Linux network stack, this is the entry point of the `softirq` for receive network processing.

The network card driver, which receives packets from the NIC, is modified to enqueue the received packets into a special producer-consumer 'aggregation queue'. The Receive Aggregation routine, running in `softirq` context, 'consumes' the packets dropped into the queue and processes them for aggregation. The 'aggregation queue' is a per-CPU queue, and is implemented in a lock-free manner. Thus, there is no locking overhead incurred for accessing this queue concurrently between different CPUs, or between the `interrupt` context and the `softirq` context.

The packets dropped into the aggregation queue by the NIC driver are 'raw' packets, i.e., they are not encapsulated in the Linux socket buffer metadata structure, `sk_buff`. The reason for this, as discussed in section 2.2, is that memory management of `sk_buffs` is a significant part of the buffer management overhead of network packets. We avoid this overhead by allocating the `sk_buff` only for the final aggregated packet, in the Receive Aggregation routine. For Linux drivers, this also allows us to avoid the MAC header processing of network packets in the driver, which is moved to the Receive Aggregation routine.

Packets are consumed from the aggregation queue by the aggregation routine and are hashed into a small lookup table, which maintains a set of partially aggregated TCP packets. If the new network TCP packet 'matches' a previously hashed packet, i.e., it can be coalesced with this packet (based on the conditions described in 3.1), then the two are aggregated. Otherwise, the partially aggregated packet is delivered to the network stack, and the new packet is saved in the lookup table.

Packets delivered to the network stack are processed synchronously, and thus control returns to the aggregation routine only when the network stack is idle. Thus, in order to remain work-conserving, whenever the aggregation routine runs out of network packets to process (i.e, the aggregation queue is empty), it immediately clears out all partially aggregated packets in the lookup table, and delivers them to the network stack. This ensures that packets do not remain waiting for aggregation, while the network stack is idle.

3.6 Correctness

Receive Aggregation is done only for a restricted set of in-order TCP packets which the TCP layer 'expects' while it is in error-free mode of operation. Any TCP packet requiring special handling by the TCP layer, off the common path, is passed on to the stack without aggregation, and thus is handled correctly by the TCP layer. Thus, all the error-handling and special case handling of packets in the TCP layer works correctly. We give a few examples below:

1. Duplicate or Out-of-order packets: Since these packets are not in correct sequence (by TCP sequence number), they are not aggregated and are handled directly by the TCP layer.

2. Selective TCP ACKs: Since TCP options other than the timestamp option are not handled by aggregation, TCP packets with selective ACKs are passed unmodified.

3. Duplicate ACKS: A duplicate ACK packet does not contain data in its payload. Since pure ACKs are never aggregated, these are handled correctly by the TCP layer.

We now explain why using timestamps from only the last TCP fragment in the aggregated packet does not result in lack of precision. At gigabit transmit rate, a single TCP sender machine can transmit packets at the rate of roughly 81,000 packets per second. The precision of the timestamp value itself, however, is typically 10 ms (if the system uses a 100 Hz clock), or 1 ms at best (with a 1000 HZ clock). Thus, roughly every 80 consecutive packets transmitted by a sender are expected to have the same timestamp to begin with. Since Receive Aggregation coalesces together packets which arrive very close to each other in time, we expect these packets to already have the same timestamp value.

4 Acknowledgment Offload

Our second optimization for reducing the per-packet overhead of receive processing is Acknowledgment Offload. Acknowledgment Offload reduces the number of TCP ACK packets that need to be processed on the transmit path of receive processing, and thus reduces the overall per-packet overhead.

TCP acknowledgment packets constitute a significant part of the overhead of TCP receive processing. This is because, in the TCP protocol, one TCP ACK packet must be generated for every two full TCP packets received from the network. Thus, TCP ACK packets constitute at least a third of the total number of packets processed by the network stack. Since the overhead of the network stack is predominantly per-packet, reducing the TCP ACK transmission overhead is essential for achieving good receive performance.

4.1 Basic Idea

Acknowledgment Offload allows the TCP layer to combine together the transmission of consecutive TCP ACK packets of the same TCP connection into a single 'template' ACK packet.

To transmit the successive TCP ACK packets, the TCP layer creates a 'template' TCP ACK packet representing the individual ACK packets. The template ACK packet is sent down the network stack like a regular TCP packet. On reaching the NIC driver (or a proxy for the driver), the individual TCP ACK packets are re-generated from the template ACK packet, and are sent out on the network.

4.2 Template ACK packet

The template ACK packet for a sequence of consecutive ACKs is represented by the first ACK packet in the sequence, along with the ACK sequence numbers for the subsequent ACK packets, which is stored in the template packet's metadata structure (`sk_buff` in Linux).

The TCP and IP headers of the successive ACK packets of a TCP connection share most of the fields of the header. In particular, only the ACK sequence number and the IP checksum field differ between the successive packets. (This is assuming they are generated sufficiently close in time, so that the TCP timestamps are identical). Thus, the information present in the template ACK packet is sufficient to generate the individual ACK packets in the sequence.

The NIC driver is modified to handle the template ACK packet differently. The template packet is not transmitted on the network directly. Instead, the driver makes the required number of copies for the network ACK packets. It then rewrites the ACK sequence number for the network packets, recomputes the TCP checksum, and transmits the sequence of TCP ACK packets on the NIC.

4.3 When it is used

Acknowledgment Offload is preferably used in conjunction with the Receive Aggregation optimization. This is because, in a conventional TCP stack, TCP ACK packets are generated and transmitted synchronously in response to received TCP packets (except for delayed TCP ACKs). Since the received TCP packets are also processed by the TCP stack synchronously, the TCP layer does not generate opportunities to batch together the generation of successive ACK packets.

However, with Receive Aggregation, an aggregated TCP packet effectively delivers multiple network TCP packets to the TCP layer simultaneously. This provides the TCP layer an opportunity to generate a sequence of consecutive TCP ACK packets simultaneously, and at this point, it can make use of Acknowledgment Offload to transmit the ACK packets.

5 Evaluation

We have implemented Receive Aggregation and Acknowledgment Offload in a stock 2.6.16.34 Linux kernel, and in the Xen VMM version 3.0.4 running Linux 2.6.16.38 guest operating systems.

We first evaluate the performance benefits of the receive optimizations for three systems: a uniprocessor

Linux system, an SMP Linux system, and a Linux guest operating system running on the Xen VMM. Next, we experimentally determine a good cut-off value for the Aggregation Limit. In the third set of experiments, we demonstrate the scalability of our system as we increase the number of concurrent receive processing connections. Finally, we demonstrate that our optimizations do not affect the performance of latency-sensitive workloads.

5.1 Performance Benefits

We use a receive microbenchmark to evaluate the TCP receive performance of the system under test. This microbenchmark is similar to the netperf [1] TCP streaming benchmark and measures the maximum TCP receive throughput which can be achieved over a single TCP connection.

The server machine used for our experiments is a 3.0 GHz Intel Xeon machine, with 800 MHz FSB and 512 MB of DDR2-400 memory. The machine is equipped with five Intel Pro1000 Gigabit Ethernet cards, fitted in 133 MHz, 64 bit PCI-X slots. We run one instance of the microbenchmark for each network card. The 'receiver' end of each microbenchmark instance is run on the server machine and the 'sender' end is run on another client machine, which is connected to the server machine through one of the Gigabit NICs. The sender continuously sends data to the receiver at the maximum possible rate, for the duration of the experiment (60s). The final throughput metric reported is the sum of the receive throughput achieved by all receiver instances.

Overall Results

Figure 7 shows the overall performance benefit of using Receive Aggregation and Acknowledgment Offload in the three systems. The figure compares the receive performance of the three systems (throughput, in Mb/s) with and without the use of the receive optimizations. The 'Linux UP' histograms show the performance for the uniprocessor Linux system, 'Linux SMP', for the SMP Linux system, and 'Xen', for the Linux guest operating system running on the Xen VMM.

The performance results for the three systems are as follows.

For the uniprocessor Linux system, the unmodified (Original) Linux TCP stack reaches full CPU saturation at a throughput of 3452 Mb/s. With the use of the receive optimizations, the system (Optimized) is able to saturate all the five Gigabit network links, to reach a throughput of 4660 Mb/s. The CPU is still not fully saturated at this point and is at 93% utilization. The system is thus constrained by the number of NICs, and with more NICs, it can theoretically reach a (CPU-scaled) throughput of 5050 Mb/s. The performance gain of the system is thus 35% in absolute units and 45% in CPU-scaled units.

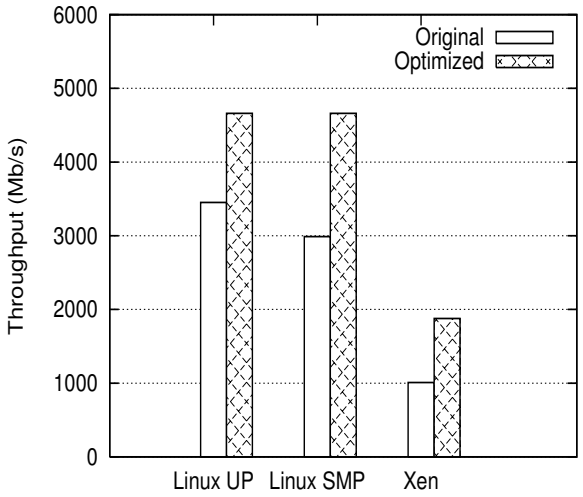

Figure 7: Overall Performance Improvement

For the SMP Linux system, the performance of the Original system is 2988 Mb/s, whereas the optimized system is able to saturate all five NICs to reach a throughput of 4660 Mb/s. As in the uniprocessor case, the optimized SMP system is still not CPU saturated and is at 93% CPU utilization. Thus, the performance gain in the SMP system is 55% in absolute terms and 67% in CPU-scaled units.

For the Linux guest operating system running on Xen, the unoptimized (Original) system reaches full CPU saturation with a throughput of only 1088 Mb/s. With the receive optimizations, the throughput is improved to 1877 Mb/s, which is 86% higher than the baseline performance.

The contribution of Acknowledgment Offload to the above performance improvements is non-trivial. Using just Receive Aggregation without Acknowledgment Offload, the performance improvement to the three configurations is, respectively, 26%, 36% and 45%, with CPU utilization reaching 100% in all three cases.

Analysis of the Results

We can better understand the performance benefits of the receive optimizations by comparing the overhead profiles of the network stack in the three configurations, with and without the use of the optimizations.

Figure 8 compares the performance overhead of the network stack for the uniprocessor Linux configuration, and shows the breakdown of the CPU cycles incurred per packet on the receive path. In addition to the different

per-packet and per-byte categories discussed in section 2, there is a new category `aggr`, which measures the overhead of doing Receive Aggregation. The overhead of Acknowledgment Offload itself is included as part of the device driver overhead.

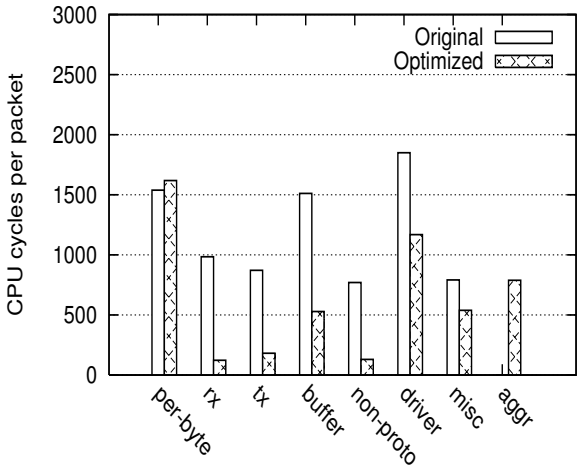

Figure 8: Receive processing overheads (UP)

Receive Aggregation and Acknowledgment Offload effectively reduce the number of packets processed in the network stack by a factor of up to 20, which is the Aggregation Limit on our system. This greatly reduces the overhead of all the per-packet components in the network stack. The total overhead of all per-packet components (`rx`, `tx`, `buffer` and `non-proto`) is reduced by factor of 4.3.

The main increase in overhead for the optimized network stack is the Receive Aggregation function itself (`aggr`). We note that the bulk of the overhead incurred for Receive Aggregation (789 cycles/packet) is due to the compulsory cache miss which is incurred in the early demultiplexing of the packet header. Since the device driver itself does not perform any MAC header processing in the optimized network stack, its overhead is reduced by 681 cycles/packet, since it avoids the compulsory cache miss.

Figure 9 shows the receive processing overhead for the optimized and unoptimized network stack in the SMP Linux configuration.

The overall trends in this figure are similar to those for the uniprocessor configuration. As in the uniprocessor case, the overhead of the per-packet routines, `rx`, `tx`, `buffer` and `non-proto` is greatly reduced. With the receive optimizations, the total overhead of all the per-packet components (`rx`, `tx`, `buffer` and `non-proto`) in the network stack is reduced by a factor of 5.5.

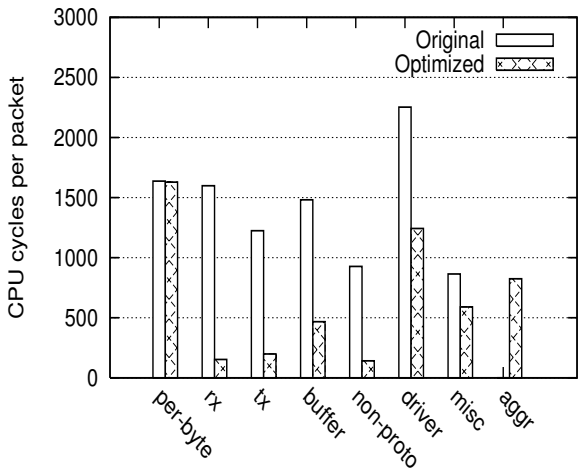

Figure 9: Receive processing overheads (SMP)

In the original SMP configuration, there is an increase in the overhead of the per-packet TCP routines relative to the uniprocessor case, because of the locking and synchronization overhead in the TCP stack. In the optimized network stack, both Receive Aggregation and Acknowledgment Offload are implemented in a CPU-local manner, and thus do not incur any additional synchronization overheads.

Finally, figure 10 shows the breakdown of the receive processing overhead with receive optimizations in the Linux guest operating system.

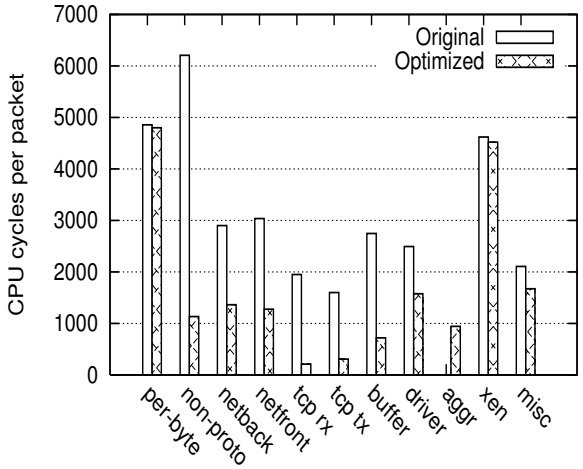

Figure 10: Receive processing overheads (Xen)

In the guest Linux configuration, the total overhead of the per-packet routines (`non-proto`, `netback`, `netfront`, `tcp rx`, `tcp tx` and `buffer`) in the

network virtualization stack is reduced by a factor of 3.7 with the use of the receive optimizations. The greatest visible reduction is in the overhead of the `non-proto` routines, which includes the bridging and netfilter routines in the driver domain and the guest domain. The overheads of the `netfront` and `netback` paravirtual drivers are reduced to a lesser extent, primarily because they incur a per-TCP fragment overhead instead of a purely per-packet overhead. Other per-packet components, such as the TCP receive and transmit routines (`TCP rx` and `TCP rx`) and buffer management (`buffer`) show similar reduction in overhead as in the native Linux configurations.

The overhead of Receive Aggregation (`aggr`) itself is small compared to the other overheads.

5.2 Choosing the Aggregation Limit

The performance benefit of Receive Aggregation is proportional to the number of TCP packets which are combined to create the aggregated host TCP packet. A greater degree of aggregation results in a greater reduction in the per-packet overhead. However, beyond a limit, packet aggregation does not yield further benefits. We determine a good cut-off value for this Aggregation Limit experimentally.

Figure 11 shows the total CPU execution overhead (in CPU cycles per packet) incurred for receive processing in a uniprocessor Linux system, as a function of Aggregation Limit.

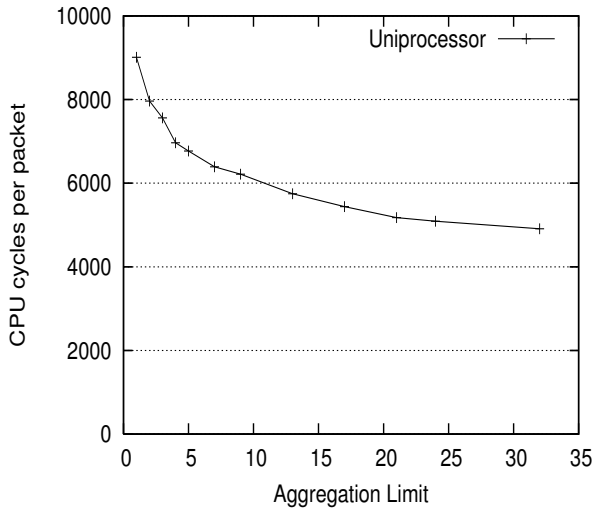

Figure 11: CPU overhead vs. Aggregation Limit

Increasing the Aggregation Limit initially yields a sharp reduction in the CPU processing overhead of packets. The figure shows that most of the benefits of Receive Aggregation can be achieved with a relatively small Aggregation Limit. We choose a value of 20 for the Aggregation Limit as it can be seen that additional aggregation does not yield any substantial improvement.

The Aggregation Limit measured above can also be derived analytically, since it only depends on the percentage overhead of the per-packet operations whose overhead can be scaled down by aggregation. For instance, if `x`% of the overhead is constant, and `y`% is the per-packet overhead that can be reduced by aggregation (with `x + y = 100`), then using an aggregation factor of `k` should reduce the system CPU utilization from `x + y` to `x + y/k`. Figure 11 appears to match the plot of `x + y/k` as a function of `k` fairly well. This gives us confidence that the Aggregation Limit chosen is quite robust and not arbitrary, and it will hold across a number of different systems.

5.3 Scalability

The previous sections demonstrate the performance benefits of our optimizations when the workload consists of a small number of high-volume TCP connections. We now evaluate how the optimizations scale as we increase the number of concurrent TCP connections receiving data.

The benchmark we use is a multi-threaded version of the receive microbenchmark. We create a number of receiver threads, each of the threads running the receiver microbenchmark and connected to a different sender process. We measure the cumulative receive throughput as a function of the number of receive connections.

Figure 12 shows how the system scales as a function of the number of connections, both in the original and the optimized system. The figure compares a baseline 2.6.16.34 Linux SMP system (`O`riginal), with the optimized Linux system (`O`ptimized).

The figure shows that our optimizations scale very well even as we increase the number of concurrent connections to 400, with the optimized system performing 40% better than the baseline system, at 400 connections. This demonstrates that Receive Aggregation is effective in reducing the number of packets even in the presence of concurrency.

5.4 Impact on Latency Sensitive Workloads

We use the netperf [1] TCP Request/Response benchmark to evaluate the impact of the receive optimizations on the latency of packet processing.

This benchmark measures the interactive request-response performance of a client and server program connected by a TCP connection. The client sends the server a one-byte 'request', and waits for a one-byte

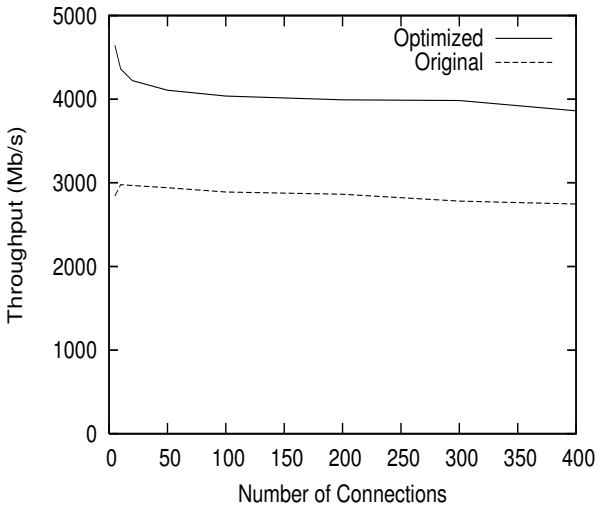

Figure 12: Scalability

	Requests/sec (Original)	(Optimized)
Linux UP	7874	7894
Linux SMP	7970	7985
Xen	6965	6953

Table 1: Impact of Receive Optimizations on Latency

'response' from the server. On receiving the response, the client immediately sends another request. The benchmark measures the maximum request-response rate achieved between the client and the server.

Table 1 compares the performance of the three systems on the TCP Request/Response benchmark. The table shows that our receive optimizations have no noticeable impact on the latency of packet processing in the network stack.

This is because of the work-conserving nature of Receive Aggregation. Since there is only one network packet to process at a time, no packet aggregation is done, and the packet is passed on to the network stack immediately to prevent it from being idle.

5.5 Discussion

The performance results presented in this section were acheived in a LAN environment, where the low network latencies and small inter-packet delays allow Receive Aggregation to effectively coalesce multiple consecutive TCP packets. The important condition for Receive Aggregation to work effectively is to have a sufficient number of consecutive TCP packets received within short interval of each other.

One example of a real-world situation where Receive Aggregation is applicable is a Storage Area Network (SAN) using iSCSI, where storage servers have high bandwidth processing requirements for transferring (including receiving) large files. In general, data intensive workloads running in a LAN environment would gain the most from these optimizations.

Under other network conditions, the performance benefits of our optimizations may vary, depending on the degree of aggregation possible. However, the overall performance will never get worse than the original system. We verified this by setting the Aggregation Limit to one in our LAN experiments, which measures the overhead of our system in the absence of any aggregation. We observed no degradation in the performance relative to the baseline.

6 Related Work

Initial analysis of TCP performance [3] identified the per-byte data touching operations to be the major source of overhead for TCP. This led to the development of a number of techniques for avoiding data copy, both in software [15] [10], and hardware [13, 11]. Techniques such as zero-copy transmit and hardware checksum offload have now become common in modern network cards [12, 4, 6].

Later work [9] identified the per-packet overhead as the dominant source of overhead for real-world workloads, which are dominated by small message sizes. This led to the development of offloading techniques for reducing per-packet overheads, such as TCP segmentation offload.

Recently, some high end network cards have started providing more complex offload support for TCP receive processing, such as Large Receive Offload (LRO) in Neterion NICs [8]. The idea of LRO is similar to that of Receive Aggregation, except that it is performed in the NIC, and thus it can reduce the per-packet overhead incurred in the network driver. However, a pure-software approach such as Receive Aggregation is much more generic, and can yield much of the benefit of packet aggregation in a hardware independent manner. Additionally, the Neterion NIC does not support Acknowledgment Offload, and thus does not offer support for reducing the overhead on the ACK transmit path.

Jumbo frames, which allow the ethernet MTU size to be set to 9000 bytes, can also effectively help reduce the per-packet overheads for bulk data transfers. However, they require the whole LAN network to be upgraded to use the same MTU size. Receive Aggregation and Acknowledgment Offload are effective at improving the network stack performance irrespective of the network MTU size or networking hardware used.

Receive Aggregation requires TCP packets to be demultiplexed early on in the network stack. Similar early

demultiplexing mechanisms have been explored in the context of resource accounting in Lazy Receive Processing (LRP) [5]. LRP, however, does not yield any performance improvements.

The idea of Receive Aggregation is also similar to the idea of interrupt throttling supported by many network cards. Since interrupt processing is expensive, interrupt throttling prevents Operating Systems from spending too much time in processing interrupts [14]. Similarly, Receive Aggregation reduces the CPU overhead of TCP receive processing by reducing the number of host TCP packets that the network stack has to process.

7 Conclusions

In this paper, we showed that architectural trends in the evolution of microprocessors have shifted the dominant source of overhead in TCP receive processing from per-byte operations, such as data copy and checksumming, to the per-packet operations. Motivated by this architectural trend, we presented two optimizations to receive side TCP processing, Receive Aggregation and Acknowledgment Offload, which reduce its per-packet overhead. These optimizations result in significant improvements in the performance of TCP receive processing in native Linux (by 45-67%), and in virtual Linux guest operating systems running on the Xen VMM (by 86%).

8 Acknowledgments

We would like to thank Emmanuel Cecchet, Olivier Crameri, Simon Schubert, Nikola Knezevic and Katerina Argyraki for discussions and useful feedback. This work was supported in part by the Swiss National Science Foundation grant number 200021-111900.

References

[1] The netperf benchmark. `http://www.netperf.org/netperf/NetperfPage.html`.

[2] Oprofile. `http://oprofile.sourceforge.net`.

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

[4] Intel Corporation. Small packet traffic performance optimization for 8255x and 8254x Ethernet Controllers. Technical Report application Note (AP-453), Sept 2003.

[5] Peter Druschel and Gaurav Banga. Lazy Receive Processing (LRP): A Network Subsystem Architecture for Server Systems. In *OSDI*, 1996.

[6] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. Tcp performance re-visited. In *International Symposium on Performance Analysis of Systems and Software, IPASS*, Austin, TX, 2003.

[7] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, Oct 2004.

[8] Leonid Grossman. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In *Ottawa Linux Symposium*, Ottawa, 2005.

[9] Jonathan Kay and Joseph Pasquale. Profiling and Reducing Processing Overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4(6):817–828, December 1996.

[10] H. Keng and J. Chu. Zero-copy TCP in Solaris. In *USENIX 1996 Annual Technical Conference*, 1996.

[11] K. Kleinpaste, P. Steenkiste, and B. Zill. Software Support for Outboard Buffering and Checksumming. In *ACM SIGCOMM Symposium*, 1995.

[12] Srihari Makineni and Ravi Iyer. Architectural characterization of TCP/IP packet processing on the Pentium M microprocessor. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[13] Dave Minturn, Greg Regnier, Jon Krueger, Ravishankar Iyer, and Srihari Makineni. Addressing TCP/IP Processing Challenges Using the IA and IXP Processors. *Intel Technology Journal*, November 2003.

[14] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt drivern kernel. In *ACM Transactions on Computer Systems*, 1997.

[15] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000.