Deferred Segmentation for Wire-Speed Transmission of Large TCP Frames over Standard GbE Networks[1]

Hrvoje Bilic[*,**] Yitzhak Birk[*] Igor Chirashnya[**] Zorik Machulsky[**]
Technion[*] – Israel Institute of Technology
{bhrvoje@tx, birk@ee}.technion.ac.il
IBM Research Laboratory in Haifa[**]
{ cherry, machulsk}@il.ibm.com

Abstract

"Deferred Segmentation" (DS) is a novel approach to enhance the TCP transmission rate of large frames over standard GbE networks. DS allows large TCP frames through the sender's TCP/IP stack, and the NIC breaks them down into TCP segments of standard Ethernet MTU size. DS doubles TCP/IP performance, pushing the TCP transmission rate to the wire speed and/or reducing the host CPU load.

Unlike when the protocol stack is split between the host CPU and the NIC, or is offloaded to the NIC, DS requires, at most, a small NIC development effort and no modification of legacy applications or OS TCP/IP stacks. DS need only be implemented in the sender and is transparent to the receivers. If traffic is mostly outbound and is split among multiple receivers, so doing will not reduce performance. Finally, DS can even be implemented entirely in the NIC's device driver and still improve performance.

1. Introduction

The tremendous increase in network bandwidth over the last decade has exceeded the increase in CPU power. For TCP traffic over GbE, the host CPU that executes the TCP/IP stack software has become the bottleneck, and holds effective data rates well below wire-speed. Much research has been aimed at speeding up TCP/IP traffic over Gigabit networks. Some sought to better understand the influence of various factors on TCP/IP performance [1][2]. Some put forward specific proposals.

The general idea in most past approaches has been to offload TCP/IP stack work from the CPU to the NIC, through *TCP/IP Splitting* or *TCP/IP Offloading*.

TCP/IP Splitting [3][4]. The TCP/IP functions are divided among the NIC, OS kernel, and the application: time-consuming functions are handled by the application or by the NIC, while others are left in the OS kernel. Specific schemes differ in the details of the split.

Full TCP/IP Offloading [5][6]. The entire TCP/IP protocol stack is offloaded to the network adapter. This approach entails overall packet processing by the NIC.

Unfortunately, presently available commercial GbE NICs do not implement the advanced TCP/IP splitting ideas or offloading. Most NICs only implement TCP/IP checksum calculation and interrupt-coalescing mechanisms. This difference between research and industry apparently reflects practical considerations and constraints that were neglected by the researchers: the required NIC development efforts, the need to support legacy applications and operating systems, and the complexity, time, and cost of extensively modifying the OS TCP/IP stack.

The TCP/IP stack processing overhead is composed of per-connection, per-packet, and per-byte overhead. Per-connection overhead can only be reduced by decreasing the number of packets exchanged during connection establishment and termination. Various copy avoidance and checksum techniques reduce the per-byte overhead; per-packet overhead has remained unchanged.

In an attempt to reduce per-packet processing, Alteon Inc. defined Jumbo Frames [7] of up to 9 KB. While the potential benefit is clear, this solution has important shortcomings: it is not defined in the IEEE standards for GbE and Ethernet networks; it requires the hardware along the path and at the other end of the connection to support such 9 KB packets; it increases the likelihood of congestion along the path due to the larger switch and router buffer space occupied by the larger packets; the

[1] The core of this work is the Tech. M.Sc. research thesis of H. Bilic. Implementation and measurements were carried out at IBM in collaboration with IBM staff. Bilic is currently with Galileo Technologies, *billy@galileo.co.il*.

0-7695-1357-3/01 $10.00 © 2001 IEEE

maximum permissible packet size represents a trade-off between host overhead considerations and those of the network infrastructure. Finally, the maximum TCP frame size handled by the TCP/IP stack is 9 KB. Moreover, this maximum size cannot be substantially increased because of CRC limitations. An alternative approach that can be taken by OS vendors is to provide a modified TCP/IP stack with the ability to delegate TCP segmentation to other entities.

Our Goal has been to find ways of reducing the per-packet overhead and thus the host CPU load. Pushing TCP/IP throughput to wire-speed for GbE even with current "PC-grade" CPUs is a motivating goal. We constrained our schemes to require no changes to applications or the operating system (OS) TCP/IP stack, and to require, at most, minor changes to the NIC. Our focus is on the transfer of large TCP frames. Our proposed scheme, *Deferred Segmentation*, is the subject of this paper.

The target scenario for Deferred Segmentation (DS) comprises high-end hosts (servers) that primarily transmit TCP data to different recipients (clients). The servers are connected to GbE networks, but the recipients may even be connected to slower network segments. Examples include video, file, web servers, etc.

The remainder of the paper is organized as follows. Section 2 presents Deferred Segmentation in the context of the server-to-clients environment and Section 3 evaluates the method. Section 4 offers concluding remarks.

2. Deferred Segmentation

The main idea of Deferred TCP Segmentation (DS) is to create large *TCP frames* (up to 64 KB) and allow them to be passed through all the layers of the TCP/IP protocol stack. A frame is then spliced by the NIC into *TCP segments* of standard Ethernet MTU size (1.5 KB) for transmission. This improves performance while overcoming the main shortcomings of the Jumbo Frame approach.

DS need only be implemented on the sender side and operates correctly with standard, unaware receivers. Focusing on this scenario, we discuss only the sender side in this section.

DS in the sender comprises three elements: creation of large TCP frames by the TCP/IP stack, their subsequent segmentation into MTU-size TCP segments by the NIC, and the proper handling of side effects.

2.1. Creation of Large Frames in the TCP Stack

Maximum Segment Size (MSS) values are exchanged between the TCP connection peers in SYN packets. The maximum transfer unit (MTU) is reported to the TCP/IP stack by the NIC's device driver.

The maximum permissible size of a TCP frame is the smaller of 1) the MSS value received from the remote peer and 2) the local MTU. In order to cause the TCP/IP stack to generate large TCP frames regardless of the actual MTU and MSS values, we "spoof" them.

MTU Spoofing. The device driver reports a large (fake) MTU value to the TCP/IP stack.

MSS Spoofing. The NIC snoops received TCP SYN packets and modifies the MSS field. The TCP checksum is adjusted. The spoofed MSS value is forwarded to the host's TCP/IP stack.

2.2. TCP Segmentation by the NIC

The NIC receives a large TCP frame from the TCP/IP stack and uses the frame's IP and TCP headers as templates for generating TCP segment headers for the MTU-size segments that it creates.

The TCP peers agree on the TCP options supported. The NIC that implements the TCP segmentation does not concern itself with most of the options, allowing the "real" TCP layer to carry out the negotiations. Nonetheless, the NIC must disable those options that are not supported, yet must be applied to every (small) TCP segment. The NIC participates in this process by snooping (and modifying) SYN packets.

Finally, it should be noted that this segmentation is completely transparent to the TCP receive peer host. The TCP segmentation algorithm appears in the Appendix.

2.3. Side Effects

Checksum Recalculation. The unmodified sender TCP/IP stack calculates the checksum for the entire TCP frame. The NIC recalculates the checksum for every TCP segment.

Ack Coalescing. The receiver of TCP segments is unaware of the size of the original large frame and Acks each received segment. The sender's TCP layer, on the other hand, is unaware of the subsequent segmentation. The result is a mismatch in the number of TCP segments sent/received. This mismatch does not pose correctness problems. However, 1) the TCP congestion control window mechanism would exhibit abnormal behavior, and 2) processing the large number of Acks in the sender's TCP/IP stack creates a significant workload.

To illustrate the flow control problem, consider the sender's congestion window parameter CWND, which is increased by MSS for each received Ack. The subsequent flood of Acks could result in serious network congestion [9][10].

In order to hide this "Ack flood" from the sender TCP/IP stack, we propose Ack coalescing for segmented TCP frames. For each TCP connection, the NIC accumulates the Acks received for the individual segments

and (usually) passes on a single Ack per original TCP frame.

Our Ack coalescing mechanism offers additional benefits. For example, it decreases the number of retransmit timeouts and prevents communication pipe drain caused by duplicate Acks returned for the numerous TCP segments sent). The detailed Ack coalescing algorithm appears in the Appendix.

By combining the foregoing anomaly in TCP Congestion Control behavior with the Ack coalescing mechanism, the NIC can also take charge of congestion control and adapt it to the working environment.

3. Demonstration of Completeness and Performance Estimations

In this section, we establish the "completeness" of Deferred Segmentation by demonstrating that it really works. We also present a collection of arguments that prove it can reduce host CPU utilization and achieve GbE wire-speed with current PC-grade CPUs.

3.1. Deferred Segmentation Demonstrators

In order to establish the "completeness" of Deferred Segmentation, (i.e., that we did not neglect its side effects or overlook the need for mechanisms that support it) we implemented it and demonstrated its operation in the following two environments:

Ethernet Emulation Environment. This setup comprised two IBM emulation boards connected via PCI bus to Pentium 450 MHz hosts. The hosts ran Linux RedHat 6.0 with device drivers for the emulation boards. The boards were interconnected via Ethernet. The software running on the emulation boards' PowerPC 401 processors, emulated an Ethernet NIC. The software of one of the two boards also implemented most components of Deferred Segmentation: MSS spoofing, TCP segmentation, calculation of TCP checksum, and Ack coalescing. MTU spoofing was implemented in that board's Linux device driver.

Gigabit Ethernet Environment. This setup comprised two off-the-shelf Alteon AceNIC GbE NICs, connected via PCI bus to Pentium 450 MHz hosts running Linux RedHat 6.0 with AceNIC device drivers. The NICs were interconnected via GbE. We modified the device driver of one of the AceNICs to perform Deferred Segmentation in its entirety. The other AceNIC's driver was not modified.

Netperf, running in the two peer hosts, was used to transmit large data frames. *Tcpdump* was used for traffic monitoring.

Test 1: Ethernet Emulation Environment. We ran Netperf to generate bulk TCP traffic. Initially, we disabled all DS mechanisms. We observed small TCP frames (1.5 KB) going through the sender's TCP/IP stack. The same TCP frames were observed in the stack of the receiver side. Next, we enabled all acceleration mechanisms in the sender except for Ack coalescing. We observed the large TCP frames (up to 32 KB) going through the sender's stack, and small TCP segments being received by the receiver's TCP stack. The number of Acks received by the sender's stack was proportional to the number of small TCP segments (up to 1.5 KB) sent by the sender's emulation board. Finally, we enabled all acceleration mechanisms. We observed the same results, except that the number of Acks received by the sender's stack was proportional to the number of large TCP frames sent by the sender's TCP/IP stack.

Test 2: Gigabit Ethernet Environment. We repeated Test 1 with the standard AceNICs, with Deferred Segmentation performed entirely in the NIC device driver of the sender while using a completely standard receiver, and obtained the same results as in Test 1.

The two tests were performed solely to establish that our DS mechanisms function properly, rather than for performance evaluation.

The tests without Ack coalescing demonstrate that the sender's TCP/IP stack operates correctly even when it does not require that the number of received Acks equal the number of stack-generated TCP frames. The tests in both environments demonstrate operation of DS with unmodified TCP/IP stacks. Moreover, Test 2 demonstrates the operation of a sender that executes Deferred Segmentation with an unmodified, off-the-shelf receiver. It also demonstrates that DS can be implemented in its entirety in the NIC's device driver.

Having established that Deferred Segmentation works, we proceed to estimate the expected performance.

3.2 Performance Estimation

For "single-server – many clients" applications, processing at the receiving ends does not limit performance. Our focus is therefore on sustained effective transmission rate and on the required CPU utilization. Nonetheless, we briefly discuss latency.

Our performance estimation is composed of several components that combine to form a good estimate: 1) an assumption that the NIC can perform its standard core functions at wire-speed; 2) measurements of the performance of blocks that are not modified by us, when these blocks process large frames, and 3) an estimation of the performance of the elements that are added by DS.

NIC Core Functions. It is a trivial assumption that a NIC can perform its standard functions at GbE wire-speed.

TCP/IP Stack (Host CPU). The sender's TCP/IP stack, which was the original performance bottleneck, remains unaltered by Deferred Segmentation. However, because of MSS and MTU spoofing, we only need to

assess the rate at which the sender CPU can handle large frames. We base this assessment on [2].

In [2], the performance of the TCP/IP stack was measured for various system configurations as a function of MTU up to 32 KB. (The communication took place between two hosts that were interconnected by a Myrinet [11] network, whose MTU can be up to 32 KB.) The TCP/IP stack and all other system parameters were held constant. We refer to the results obtained from a setup in which the hosts were both DEC Monet. (Compac XP1000 Professional Workstation), with a 500 MHz Alpha 21264 CPU, DEC 21272 "Tsunami" chipset, 4 MB L2 cache, and 640 MB DRAM. Zero-copy and checksum calculation were disabled. The maximum sustained TCP transmission rate with an MTU of 8 KB or larger reached 956 Mbps. As MTU was increased above 16 KB, processor utilization gradually dropped from 100% down to 50% for an MTU of 32 KB.

Our conclusion from these measurements is that the host CPU can easily process *large* TCP frames at GbE wire-speed.

Additional Processing by NIC. Having assumed the above mentioned, we now turn our attention to the extra burden imposed on the NIC in support of DS.

GbE NICs have separate transmit (Tx) and receive (Rx) data paths, to support full duplex communication. DS may require additions to the central NIC logic, to pipeline the packet processing with the DMA engine and link layer logic.

On-the-fly checksum calculations are common in current GbE NICs, and MTU and MSS spoofing require almost no processing. We therefore only estimate the processing time required for the TCP segmentation and Ack coalescing performed by the Tx and Rx path of the sender's NIC, respectively.

Deferred TCP Segmentation (Tx Path). Original frame headers are used as templates for TCP segments and IP headers, with only a few fields requiring modification. The NIC processes the frame's header without waiting for the entire frame to arrive.

In order to sustain wire-speed, TCP segmentation must be carried out at a rate of per-segment transmission time. For large transfers, which is the case being studied, the great majority of TCP segments are 1.5 KB in size. Transmission of such a segment at 1 Gbps takes 12 μs. Non-optimized assembly level code written for this purpose requires fewer than 200 clock cycles (2 μs –at 100 MHz), well within the constraint.

Ack Processing (Rx Path). When Ack packets are received, the additional DS logic reads the packet headers, performs Ack coalescing, and occasionally forwards an Ack on to the sender's TCP/IP stack.

Ack coalescing is composed of two parts: 1) a lookup operation to determine the corresponding Ack coalescing entry, and 2) actual coalescing.

Suitable lookup algorithms (lookup by 5 tuples) and efficient lookup tables (small storage requirements) are used in packet classification engines. The time required for lookup in a table with 10,000 entries with a 100 MHz clock, is less than 2 μs (conservative). Written non-optimized Ack coalescing assembly code (excluding lookup) required fewer than 50 clock cycles (0.5 μs at 100 MHz). Consequently, even without pipelining, Ack coalescing takes less than 2.5 μs, even with as many as 10,000 open TCP connections.

The average Ack's arrival rate equals the rate at which MTU size TCP segments are transmitted (i.e., one per 12 μs.) The Ack processing time is thus well within the limits even for sub-MTU segments. Bursty arrival of Acks can be handled by rate-smoothing buffers, in conjunction with the excess mean processing power. Note also that, from the host TCP/IP stack's point of view, the Ack buffer overflow is equivalent to the loss of Acks in the network. Consequently, correctness is never compromised.

3.3. Latency

End-to-end latency is the sum of the latencies in the sending host, network, and receiving host. When compared with the legacy solution, Deferred Segmentation exhibits lower sending host latency, and equal network and receiver latencies.

4. Conclusions

This paper presents Deferred Segmentation (DS), a technique for enabling wire-speed transmission of large TCP frames over standard GbE networks. DS overcomes the well-known bottleneck, namely the host CPU that executes the TCP/IP stack code. Our focus was on server applications, so we focused on the sender side.

The key idea in DS is to actually reduce the total amount of processing required per large TCP frame, by forwarding large TCP frames through the sender's protocol stack and having the NIC break them down into small segments to comply with the standard Ethernet MTU. DS provides full compliance with TCP and Ethernet standards, requires no changes to the application or to the operating system's protocol stack, no changes to the receiving end, and at most, minor changes to the sender's NIC. In contrast to Jumbo Frames, the network can be a standard GbE.

Operability of the scheme was demonstrated with off-the-shelf receivers and we provided good estimates for the expected performance.

Proposed DS mechanisms will be implemented in the IBM GbE NIC developed by the IBM VLSI department in Haifa.

More details about DS implementation and its impact on TCP mechanisms can be found in [12].

Acknowledgments. We are grateful to the members of the VLSI Design Technologies Department at the IBM Haifa Research Lab for their assistance. In particular, to Yonit Davidson, Vadim Makhervaks, and Eitan Peri. We thank Giora Biran, Claudiu Schiller, and Tal Sostheim for their NIC architecture insights. Special thanks go to Raanan Gewirtzman for enabling this project. Finally, we thank the reviewers for their insightful comments, which are addressed in [12].

5. Appendix

TCP Segmentation Algorithm. Headers for TCP segments are generated by the NIC as follows:

Ethernet Header: no changes.
IP header:
IP total length: change to the new IP frame size.
IP header checksum: calculated by NIC.
TCP header:
TCP Sequence number: increment by bytes sent in previous TCP segments (MOD pow(2,32)).
TCP checksum: calculated by NIC.
TCP flags are generated as follows:
URG bit: set only in first TCP segment if set in large frame hdr; URG pointer: no change.
Ack bit: Set only in first TCP segment if set in large frame hdr; Ack number: no change.
PSH bit: set only in last TCP segment if set in large frame hdr;
RST bit: set only in last TCP segment if set in large frame hdr;
SYN bit: send only at connection setup (no large frames); if already set, do not change.
FIN bit: set only in last TCP segment if set in large frame hdr;

ACK Coalescing Algorithm. The information stored by the network adapter per TCP connection:

- Array of first (fsn) and last (lsn) byte number of each large frame.
- First unacknowledged sequence number (f_una). Sequence no. of the first byte sent in SYN segment.
- Number of un-acked large frames in array (num_lf).

NIC init. per-connection info at connection setup (*).
During the data exchange over TCP socket, the NIC does the following:
On Tx: NIC acts for each large TCP frame sent:

```
Store (fsn and lsn); // Store fsn & lsn for new LF
Num_lf ++;      // New LF entered into the array
```

On Rx: Per received Ack, NIC search the corresponding TCP connection and the information stored for it, and acts as follows:

```
If (no entry found) {
   pass Ack to the TCP stack ;
   exit;}
If (Ack < f_una) {
   Drop the Ack;  // It is history
   exit;}
If (Ack == f_una) {
   Pass Ack to the TCP stack;  // Duplicate Ack
   exit;}
If (Ack > f_una) {
   If (Ack ≥LSN of first LF in array) {
      Drop all LF whose Ack ≥LSN from A_LF;
      For each LF dropped do num_lf --;
      Pass the Ack to the TCP stack;
      f_una = Ack;
   }Else { // Ack < LSN of first LF in array
      If  (Ack ≤ FSN of first LF in array) {
         Pass Ack to the TCP stack;
         f_una = Ack;
      }Else{// FSN<Ack<LSN of 1st LF in array
         If (FSN > f_una) { // Ack also small seg.
         Pass to the TCP stack;
         f_una = Ack;
         }Else{//It Acks a small part of large frame
                  Drop the Ack;
                  f_una = Ack;  // Duplicate Acks }}}}
```

(*) Initialization of the Ack information per TCP connection is done at TCP connection establishment. The NIC snoops SYN packet and allocates the Ack DB when it decides to support segmentation for the specific connection. The de-allocation of the Ack information is done by snooping the FIN and RST bits.

6. References

[1] J. Kay and J. Pasquale, "Profiling and Reducing Processing Overheads in TCP/IP," IEEE/ACM Trans. Net., Vol. 4, No. 6, Dec. 1996, pp. 817-828.

[2] Andrew Gallatin, Jeff Chase and Ken Yocum – Dep. of Com. Science Duke Univ. "Trapeze/IP: TCP/IP at Near-Gigabit Speeds". 1999 USENIX Tech. Conf. (Freenix Track). Jun. 1999.

[3] Aled Edwards and Steve Muir. "Experiences Implementing a High Performance TCP in User-space". Proc. Conf. on App., Tech., 1995, Cambridge, MA, U.S. pp. 196-205.

[4] Chris Maeda, and Brian N.Bershad. "Protocol Service Decomposition for High-Performance Networking". Proc 14th ACM symp. on OS. Dec. 1993, Asheville, U.S. pp. 244-255.

[5] E. Cooper, P. Steenkiste, R. Sansom, and B. Zill. ``Protocol Implementation on the Nectar Communication Processor," SIGCOMM '90, pp. 135-143, Philadelphia, Sep. 1990. ACM.

[6 R. A. Maclean and S.E. Barvick. "An Outboard Processor for High Performance Implementation of Transport Protocols". GLOBCOM '91, pp. 1728-1732, 1991.

[7] Alteon Inc. White Paper "Extended Frame Sizes for next generation Ethernets".

[8] RFC 793 - TCP protocol, Internet Engineering Task Force.

[9] V. Jacobson, ``Congestion Avoidance and Control," Proc. SIGCOMM '88, pp. 314-329, August 1988.

[10] RFC 2581, "TCP Slow Start, Congestion Avoidance, Fast Retransmit & Fast Recovery" (1997) .

[11] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W-K Su. "Myrinet", IEEE Micro, Feb. 1995.

[12] H. Bilic and Y. Birk, "Deferred Segmentation for Efficient Transmission of Large TCP Frames over Standard GbE Networks", Tech. Rep., Electrical Engr. Dept., Technion, 2001 (in preparation).