TCP Performance Re-Visited

Annie P. Foong , Thomas R. Huff , Herbert H. Hum , Jaidev P. Patwardhan[†], Greg J. Regnier
Intel Corporation
2111 NE 25th Ave
Hillsboro, OR 97124
{annie.foong,tom.huff,herbert.hum,greg.j.regnier}@intel.com

[†]Department of Computer Science
Duke University
Durham, NC 27708
jaidev@cs.duke.edu

Abstract

Detailed measurements and analyses for the Linux-2.4 TCP stack on current adapters and processors are presented. We describe the impact of CPU scaling and memory bus loading on TCP performance. As CPU speeds outstrip I/O and memory speeds, many generally accepted notions of TCP performance begin to unravel. In-depth examinations and explanations of previously held TCP performance truths are provided, and we expose cases where these assumptions and rules of thumb no longer hold in modern-day implementations. We conclude that unless major architectural changes are adopted, we would be hard-pressed to continue relying on the 1GHz/1Gbps rule of thumb.

1 Introduction

With the advent of 10 Gigabit Ethernet (GBE) and IP storage network requirements, there has been tremendous industry activity in the area of TCP/IP Offload Engines adapters (TOEs) [1, 8, 13]. The motivation behind these efforts is the recognition that the processing requirements of a 10 GBE network adapter surpasses both the CPU abilities and memory bandwidths of mainstream servers. However, the availability of production-ready TOEs and performance projections is limited at best. In order to understand the performance implications and value of TOEs, we need an update on the latest TCP implementations and performances. Although TCP performance has been widely studied in various forms [7, 10, 6], there are no performance analyses in today's context. The goal of this study is therefore two-fold:

- To confirm the validity of certain TCP performance "truths" and proposed optimizations in modern day implementations[1]
- To take measurements and analyses to a deeper level not previously taken by other researchers.

A seminal paper by Jacobson et al [7] showed that the number of instructions for TCP protocol processing itself is minimal. The main message is that the implementation of TCP, and not the protocol itself, is the limiting factor. This bodes well for the ability of TCP to scale to very high speeds. However, TCP requires substantial support from the operating system. Researchers [10] contend that such non-data touching portions are the main overheads, especially for small transfers. To alleviate the demand on the server's CPU, recent improvements in network adapters include checksum offloading, interrupt coalescing and segmentation offloading [6]. Similarly, there has also been large improvements in drivers and operating system's support of networking. Despite these optimizations over time, the distribution of TCP processing has remained fairly constant. The major overheads of TCP performance can be divided into two categories: per byte costs, primarily data-touching operations such as checksums and copies; and per packet costs, including TCP protocol processing, interrupt handling, kernel overheads (spinlocks, context switches, timers) and buffer manipulations. Furthermore, beyond CPU clock speeds, many factors affect the amount of TCP/IP throughput that a given computer can support, including memory bandwidth and capacity of

[1]Performance numbers reported in this paper do not necessarily represent the best performance available for the processors named and should not be used to compare processor performances

its I/O subsystem. More importantly, the implementations of network interfaces and the TCP/IP software play crucial roles in performance output. Of the many possibilities, we have narrowed down our study to the following:

i Focus on the bulk data transfer paths

ii Validate the 1Hz/1bps rule and interpret CPU scaling for TCP

iii Identify optimizations with a code-path analysis of one current reference implementation of TCP

iv Measure and analyze TCP processing distributions

v Identify the memory bandwidth requirements and where they are consumed

In this aspect, this paper aims to deliver a template for analyzing network processing in general. The basic methodologies used in this analysis can be extended to other workloads (e.g. micro-benchmarks for connections, SpecWeb99, etc.) or implementations (TOEs).

The rest of this paper is organized as follows: Section 2 describes the methodology and tools we use in our experiments, and highlight limitations of our approach. We take a bottom-up approach in our analysis. We explain and profile the Linux TCP transmit and receive paths in Section 3. We analyze the memory requirements of network processing in Section 4. Based on this groundwork, we present our performance results in Section 5. Our analysis also provides details and explanations where the results diverge from the expected in Section 6. We conclude with projections on what the future landscape might be for TCP in Section 7.

2 Methodology

Ideally, we would have performed all our experiments on the same set of platforms and operating systems (OS). Our original plan was to focus on Linux due to its open source. However, as we progressed in our analysis, we hit limitations imposed by hardware, measurement tools and product availability, and had to supplement our analyses with measurements on Windows®2000. We will highlight those instances here[2].

2.1 Tools

To determine the processing distribution of TCP, we used the Intel®VTune™Performance Analyzer, which allows for low-overhead sampling. The VTune™sampler interrupts the processor at specified events (e.g. every n clockticks), and records its execution context at that sample. Given enough samples, the result is a statistical profile of the ratio of time spent in a particular routine. This statistical profile identifies hotspots in the code for in-depth analysis. Important complements to VTune™include the AHA and *emon* utilities. AHA is used to sample and compare performance of two processors or frequencies, and allows us insights into CPU scaling performance at the instruction level. If two time profiles are taken at different frequencies for a given processor, the resulting analysis can identify which sections of code scale with frequency, and which may be limited by other non-scaling factors, such as memory latency and I/O. The *emon* event monitoring tool is used to collect information on the processor and chipset performance counters. This tool is necessary in getting the front-side bus (FSB) and direct memory accesses (DMA) measurements.

2.2 System Description and Limitations

The ttcp program is one micro-benchmark commonly used for bulk data transfer analysis. We use default TCP/IP settings and do not attempt to fine-tune the settings. There are several limitations to ttcp. It uses a single-stream, one-way communication. A connection is set up once between two nodes. Data is sent from the transmitter(s) to the receiver(s), reusing the same buffer space for all iterations. ttcp workload primarily characterizes bulk data transfer behavior, and must be understood in that context. We choose this simple workload because it exercises the typical and optimal TCP code path [7], and allows us to focus on understanding the network stack without application-related distractions. It also gives us the upper bound on network performance of a given transfer size. To measure total memory requirements (i.e. including DMA), we need access to counters in the chipset. At the time of data collection, *emon* was available for Windows®and only for Intel®82450NX chipsets. The available test system is a 450MHz Pentium®II processor system with the 82450NX chipset. Fortunately the measurements we are interested in are not affected by the age of this system. We use a 100Mbps network card for these experiments to better match the slower system. Our experimental setup is as follows:

	Performance/Profi les	Mem Requirement
Processor	Pentium®4	Pentium®II
Frequency	0.8GHz,2.4GHz(base)	450MHz
L2 Cache Size	512KB	2MB
Chipset	850	82450NX
GBE Adapter	Intel®Pro 1000	Intel®Pro100

Table 1. Experimental Setup

[2]Intel, Pentium are registered trademarks of Intel Corporation. Windows, Windows 2000 are registered trademarks of Microsoft Corporation. Other products and company names mentioned may be trademarks of their respective owners

3 Understanding One Reference Implementation of TCP/IP

3.1 Linux 2.4.16 Receive and Transmit fast paths

Previous work exists that traces the Linux code path from the top down (socket call) and the bottom up (driver code) [9, 13]. However, what happens in between, especially of buffers and queue management, is not as well documented. We believe that an understanding of these functions is important in characterizing TCP performance. As such, we map all the functions in the Linux TCP bulk data paths and highlight the performance-dependent details here. On a receive, there can potentially be two situations: the application issues a read before data arrives (i.e. application buffers are pre-posted), or data arrives waiting for a read. This will lead to two very different data paths (Figure 1). In the non pre-posted path, the packet is checksummed first, and then copied when the application buffers are available. In the other path, the packet undergoes a checksum and copy concurrently, a well-known optimization first proposed by Jacobson et al [7]. The correct implementation of integrated checksum and copy (csum-copy) is not trivial. In fact, researchers [6] working off Linux-2.3.99-pre8 did not see any improvement in performance with csum-copy. As we shall see in Section 4, the current version of Linux has done so successfully. Linux-2.4 makes use of three pseudo queues for this purpose: a normal receive queue, prequeue (for csum-copy), and a backlog queue (which serves as the overflow queue when the other receive queues are in use). In addition, in order for csum-copy to happen, three conditions must be met: the sequence numbers are in order, the current process is the reader waiting on the socket and the application buffer is large enough to receive data available in the socket buffer. The first 2 conditions are usually met in bulk data transfers.

The transmit (TX) path is more straightforward: csum-copy is always exercised in bulk data transfers (Figure 2). Worthy of note is that the driver maintains a memory-mapped status register which allows the TCP stack to be aware of driver resources and throttle transmits accordingly. This is done so that the driver never "drops" a packet on transmits. On successful transmission of packet(s), an interrupt occurs which reclaims buffer resources, unmap DMA and schedules the bottom half of transmit interrupt to run.

3.2 Functional breakdown of hotspots

Figures 3, 4 show the breakdown of the major components of TCP/IP processing. It was difficult to compare our TCP processing profiles with other work [7, 10, 6] due to the different viewpoints of the stack and workload used. Overall, there is general agreement along these lines: Kernel overheads, sockets and protocol processing make up the bulk of small transfers, while data touching and interrupt processing make up the bulk of large transfers. We separated protocol processing (e.g. tcp_recvmsg) from the sockets layer (e.g. sock_recvmsg, libc routines). With this view, we concur with Jacobson et al [7] that it is the implementation, and not the protocol itself, that is the bottleneck. What we typically refer to as TCP protocol processing takes up about 7% on receives, and 10-15% on transmits. On the other hand, the sockets interface and corresponding library support can take up to 34% for small transfers. Another large portion of kernel overheads is in system_call routine, which handles all the bookkeeping required to support system calls, parameter checks and context switching. We also found the CPU requirements of the driver code itself (4%-11%) to be somewhat less than previous studies, probably due to the added sophistication in driver code. E.g. instead of allocating buffers on the receipt of every packet, and increasing the time spent in interrupt, buffers are now pre-allocated, and continuously monitored for replenishment. Most NICs also support interrupt coalescing on receive. However, driver and interrupt processing still make up a large part of processing for large transfers. Not surprisingly, data-touching processing, i.e., checksums and copies, increases with transfer size. Data-touching operations of large transfers take up more than one-third of processing. We observed that a receive of 64B in ttcp exercises the non pre-posted buffers path, since data arrives much faster than the application can issue read(). This explains the relatively large percentage of processing (13%) needed in receiving a small payload of 64B. As the transfer size increase, the integrated checksum and copy path is exercised more frequently. In the current socket model, there is no way to control pre-posting of buffers.

4 Understanding TCP/IP Memory Requirements

At a high level, it is generally accepted that the data path of any network transfers involves three loads on memory. On a receive, the adapter DMAs data into the receiving packet buffers, and the CPU reads the data and writes it to the final application buffers. The reverse happens on transmits. We want to get experimental evidence of such a view, as well as quantify the effects of TCP control traffic and caching on memory bus. In reality, we find that memory loads are highly dependent on the application in question, buffer sizes and caching behavior. Our goal in this section is to provide a detailed walk-through of memory access patterns of an application, and account for cache coherency protocols. We compare our theoretical expectations to measured values to validate the analysis.

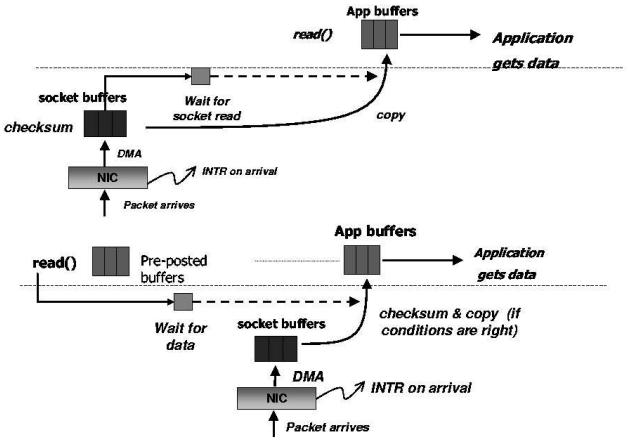

Figure 1. Receive Path: Non-Preposted Buffers(top); Pre-Posted Buffers(bottom)

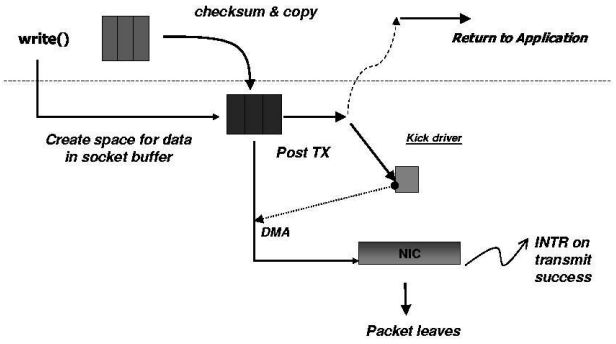

Figure 2. Transmit Data Path

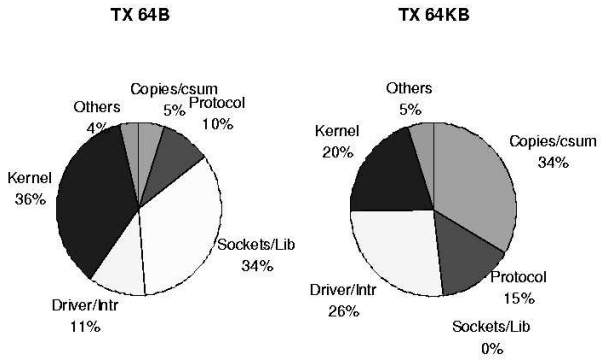

Figure 3. TCP Processing Distribution (TX)(as % of non-idle time)

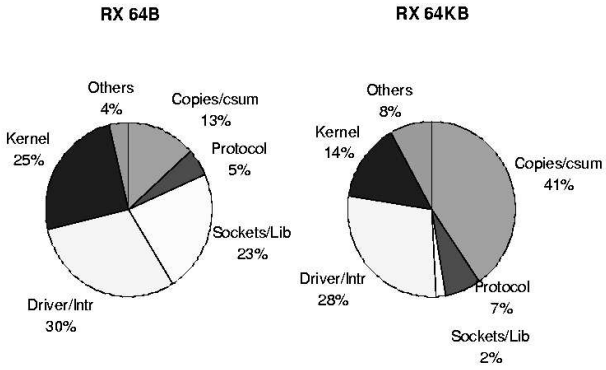

Figure 4. TCP Processing Distribution (RX)(as % of non-idle time)

4.1 Transmit Path

Total memory traffic is made up of front side bus (FSB) reads/writes and DMA reads/writes. Figure 5 shows memory traffic during a transmit. Figure 6 gives the measured FSB traffic and Figure 7 gives the DMA traffic. The numbers referred to in this analysis represent the ratio of memory to network traffic (e.g. A ratio of 2 implies that 2 bytes are generated on the memory bus per byte of network traffic). The differences between expected and measured traffic is due to TCP control traffic, retransmissions, contexts structures, cache invalidations and snoops. The 64B data point is an exception as TCP control packets have a significant effect at small transfer sizes. We use an L2 cache size of 2MB.

In the case where transfer size $<$ L2 cache, the memory transactions are as follows:

CPU reads the application buffer (1): ttcp is written such that the application buffer is reused on every iteration. The buffer is brought into cache once, and stays valid. Subsequent reading of this buffer gets its copy from L2. We expect near zero FSB reads [measured=0.1].

CPU writes to socket buffer (2): We expect the socket buffer (typically 64KB) to fit in L2. Again, it is brought into the cache once on initialization, and stays in cache throughout. On writing to the socket buffer, the copy in host memory is updated according to the normal cache coherence protocol (i.e, the written cache line is evicted at a later point or is returned to memory upon a snoop from I/O).

NIC DMAs from the socket buffer (3): This generates one DMA read [measured=1.3]. The Memory IO Controller (MIOC) takes the opportunity to capture (snarf) the outgoing DMA data during an implicit write-back to host memory. The number of loads expected on the memory bus is 2

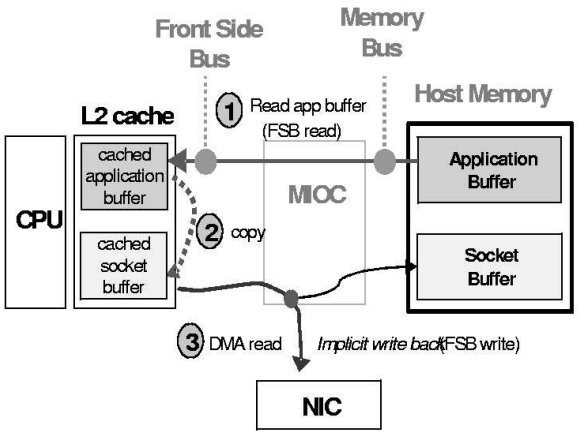

Figure 5. S/W transmit path

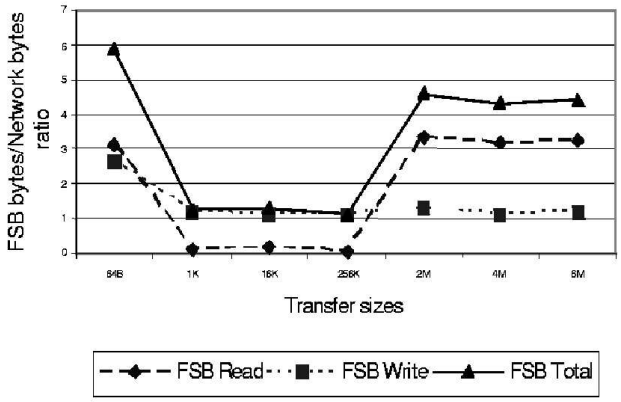

Figure 6. Measured FSB Traffic on TX

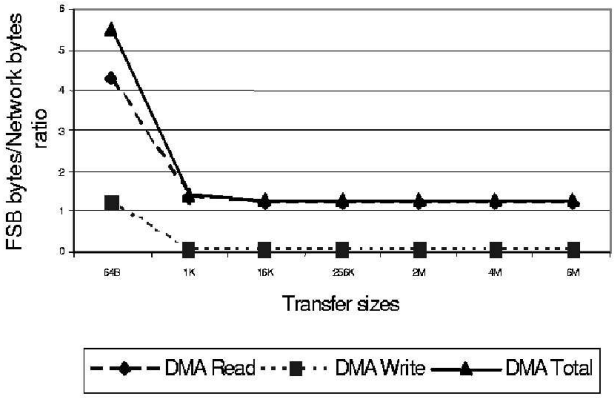

Figure 7. Measured DMA Traffic on TX

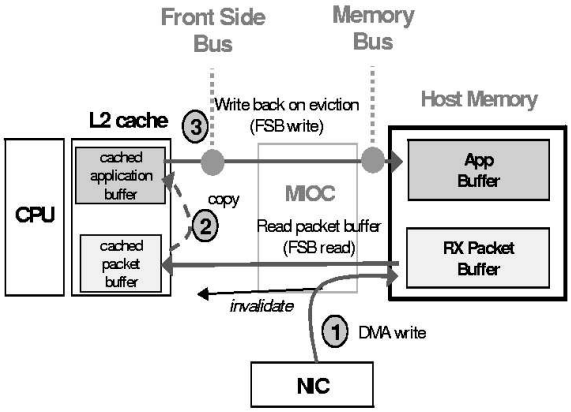

Figure 8. Receive Path

[measured=2.5].

In the case where transfer size > L2 cache:

CPU reads application buffer (1): The application buffer is now larger than L2 and is brought into cache on every iteration. We expect at least one FSB read. In addition, ttcp attempts to emulate real applications by writing a pattern into the application buffer before transmission. In other words, an additional fetch from memory is done. Contexts are also continuously being brought in and evicted from L2. We had originally thought that this will increase the memory load on the FSB. Eviction of modified contexts would bring about an increase in FSB writes (due to write backs). However, such was not the case. Due to the large payload bytes involved at the larger transfer sizes, contexts evictions and context-related write-backs do not play a significant role in FSB traffic.

CPU writes to socket buffer (2): Depending on eviction policies, the socket buffer is most likely not in cache. This would require that a copy be brought in on every iteration. This gives rise to an extra FSB read than the no-cache case considered in the overview analysis.

NIC DMAs from socket buffer (3): same as before.

In summary, expected FSB reads = 3 [measured=3.1]; Expected FSB writes = 1 [measured=1.1]; Expected DMA reads = 1 [measured=1.3]; Expected DMA writes = 0 [measured=0.06]; Expected total load on the memory bus = 5 [measured=5.6].

4.2 Receive Path

We repeat the same analysis for the receive path. Figure 8 shows the receive path. Figure 9 shows the measured FSB traffic and Figure 10 shows the measured DMA traffic

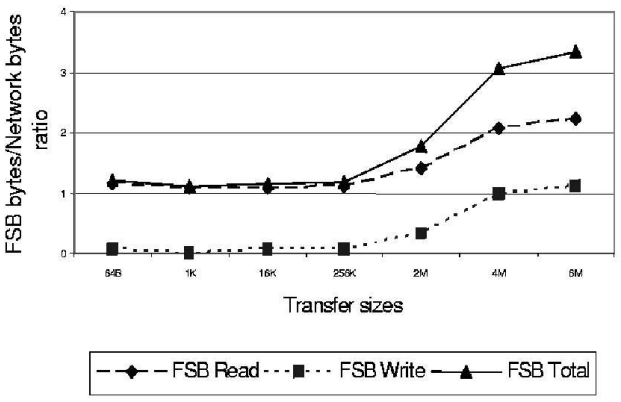

Figure 9. Measured FSB Traffic on RX

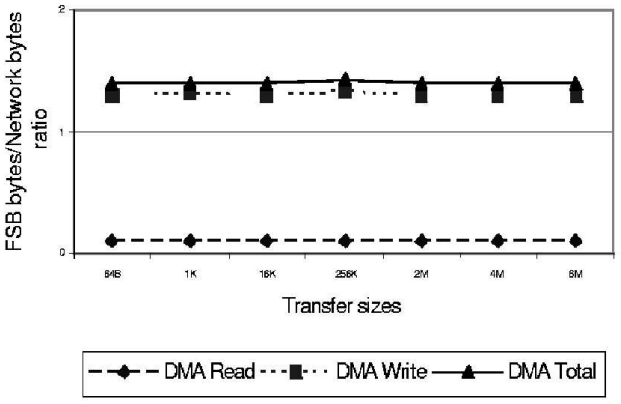

Figure 10. Measured DMA Traffic on RX

on the receive path.

For the case where transfer size $<$ L2 cache:

NIC DMAs to the packet buffer (1): The NIC DMAs data into the receive packet buffer and generates one DMA write [measured=1.3]. The cached copy of packet buffer is invalidated.

CPU reads from the packet buffer (2): For small transfer sizes, we expect the packet buffer to be in cache. However, this copy was previously invalidated by the DMA write, and a fetch from memory is needed. This generates one read on the FSB [measured=1.1].

CPU writes to the application buffer (3): As before, the application buffer is brought into cache at initialization, and remains valid. The copies occur without evictions, and should not generate any FSB write traffic [measured=0.05]. No DMA read traffic is expected [measured=0.1]. Total load expected on the memory bus is 2 [measured=2.5].

For the case where transfer size $>$ L2 cache:

(1),(2) are same as before

CPU writes to the application buffer (3): Here again, the application has to be brought into cache on every iteration. We expect at least one FSB read. Additionally, cache evictions will also force a write-back of the modified application, causing one FSB write. The net effect is slightly worse than the case where a cache is non-existent (due to extra traffic due to invalidates and cache snoops). In summary, expected FSB read = 2 [measured=2.2]; Expected FSB write = 1 [measured=1.1]; Expected DMA read 0 [measured=0.1]; Expected DMA read = 1 [measured=1.3]; Expected total load on the memory bus = 4 [measured=4.7];

Not surprisingly, there is a one to one mapping of network payload to DMA payload. This ratio stayed fairly constant regardless of transfer sizes. Extra memory accesses are incurred by control traffic and retransmits. As for the total load on memory, the accepted view of three loads on the memory bus is a rule of thumb at best. Worthy of note is the "step-function" that occurs for FSB traffic when the transfer size is the same as the processor's cache size. traffic. When transfer sizes are less than L2, the cache is large enough to accommodate all of the application buffers, the kernel's socket and packet buffers, contexts and ttcp code (minimal). There is a valid copy of application or kernel buffers in the L2 cache throughout a ttcp run. This may not be generally true. On the other hand, to completely negate the effects of the L2 cache, we have used transfer sizes of (2MB - 6MB), which is atypically large. Every iteration causes a full eviction of buffers and TCP contexts, giving rise to the sharp increase in FSB traffic. These two extremes form the upper and lower bound on memory bandwidth requirement for realistic workloads.

5 Understanding TCP/IP Performance

5.1 Validating 1GHz/1Gbps rule of thumb

The generally accepted rule of thumb is that 1bps of network link requires 1Hz of CPU processing. Figures 11, 12 give a full story of this rule of thumb. (where Hz/bps ratio = %CPU utilization * processor speed / bandwidth). It had held up remarkably well over the years, albeit only for bulk data transfer at large sizes. For smaller transfers, we found the processing requirement to be 6-7 times as expected. Moreover, the figures show that network processing is not scaling with CPU speeds. The processing needs per byte increase when going from 800MHz to 2.4GHz. This happens because as CPU speed increases, the disparity between memory and I/O latencies versus CPU speeds intensifies. The processor is held up frequently waiting for memory or I/O accesses, during which no work can be done. "Idle" time at such fine granularity is not detected by the OS, and thus is unable to context switch. Those idle times are accounted as work done and adds to CPU utilization. The

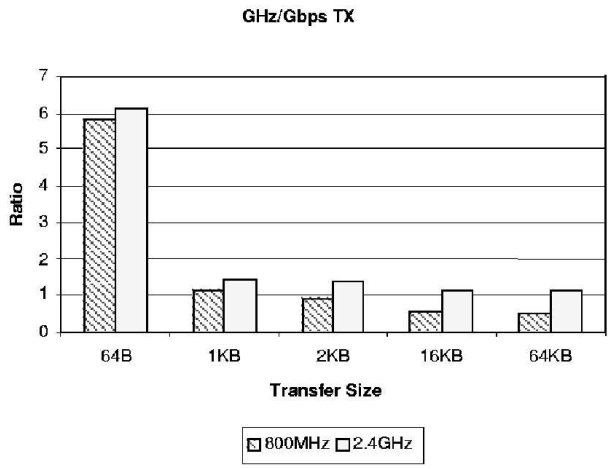

Figure 11. CPU requirements of network processing (TX)

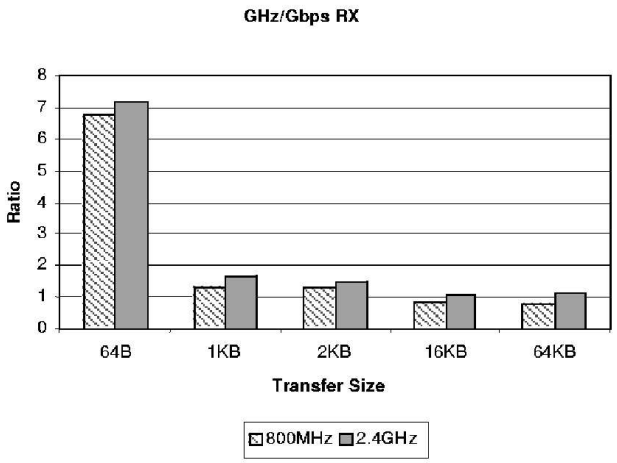

Figure 12. CPU requirements of network processing (RX)

scaling effect is more obvious at larger transfers. As seen earlier, interrupts and copies make up the largest hotspots at these transfer sizes. Interrupts incur un-cacheable (UC) I/O accesses, while copies incur memory accesses. The increase in Hz/bps is also more obvious in the transmit path due to the UC access on every packet transmission. Taking measurements at two frequency points, we found that performance is only scaling at approximately 60% of frequency. The VTune™AHA methodology helped us identify a number of opportunities for additional optimization, including: elimination of UC accesses, improved memory copy loops, and data pre-fetch tuning. Incremental optimizations may be possible, but the bottom-line remains that as CPU speeds go up, the 1Hz/bps no longer holds.

5.2 Receives versus transmits

Maximum throughput (900Mbps, data only) was achieved quite readily for the larger payload sizes (Figure 13). It is therefore safe to assume that the NICs are able to process at line speeds. It is generally accepted that receive processing is the heavier side [7]. Results at 800MHz showed that such an assumption is true . CPU requirements per byte are generally lesser for transmit than for receive. However, this is not the case at 2.4GHz (Figure 13). We believe that one possible explanation is again in the latency in UC accesses. An update of the “tail pointer” of the descriptor ring occurs on every successful transmission of a packet. This results in an un-cacheable access to memory-mapped I/O. Though CPU speeds have increased from 800MHz to 2.4GHz, the bandwidth of un-cacheable operations remained the same at 100-200MBps (the interconnect between the processor and memory controller hub-called the P4 bus-operated at a fixed frequency). This points to the requirement that P4 bus frequencies must increase proportionately with CPU speeds. The same UC accesses occur on the receive side, but the update happens only when receive resources run out, and there is a natural coalescing on the receive path. As such, I/O and memory-related latencies are far more pronounced in the transmit path than the receive path.

5.3 Checksum offload on receives

Many modern NICs support checksum offloads under the assumption TCP checksumming takes up a substantial part of the data touching processing [10, 6, 4]. Though checksum offload is generally beneficial, Figure 14 shows only a non-impressive recovery of approximately 10% of CPU cycles when checksum is offloaded.

The 64B payloads present an interesting case (Figure 14). CPU utilization is actually larger when using HW checksum. However, a larger throughput is also achieved. The expected decrease in CPU utilization is seen only in the larger payloads. Checksumming is a per byte operation, and thus has bigger effect on large payloads. Any CPU cycles reduction that can be attained by HW checksumming is expected to show itself more in the large payloads. However, there is a more subtle (opposing) behavior at play here, which is seen only with a Hz/bps performance (Figure 15) graph. Figure 15 shows gains across all transfer sizes, with the largest gain in efficiency at the 64B transfers. As we discussed in section 3, the Linux receive path differs for small and large payloads. The large transfers exercise the

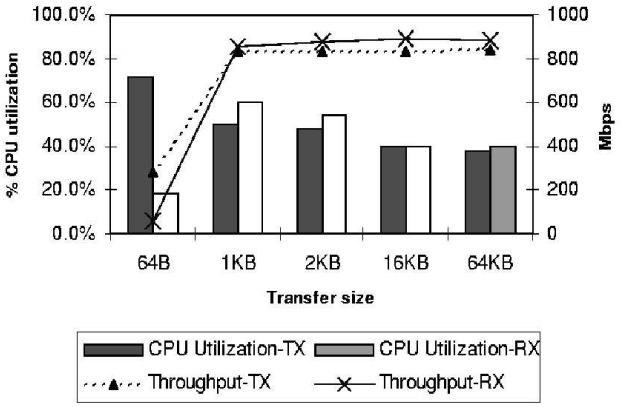

Figure 13. RX versus TX performances (at 2.4GHz)

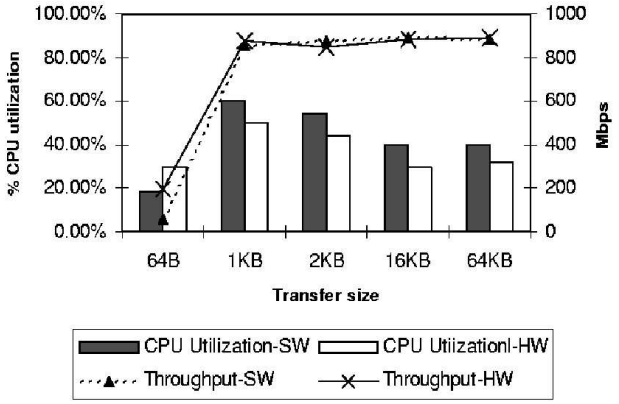

Figure 14. SW versus HW checksum (RX):Raw performance

integrated csum-copy, while the small transfers exercise the csum and then copy routines as two separate calls. If the checksum is offloaded to the NIC, the benefit manifests itself more for the small transfers.

Upon further examination of the integrated checksum routine, we noted the two forms of the operation as shown below:

```
csum-copy:                csum:
  mov   mem -> reg;         mov   mem -> reg
  adcl                      adcl
  mov   reg -> mem;         mov   reg -> mem
                          copy:
                            mov   mem -> mem
```

A breakdown of checksum (adcl) versus copy (mov) done in the csum-copy routine, shows that the copying dominates over the checksum operations by a large margin (90%) (Figure 16). Such an observation has two implications: i) that checksumming has been implemented effectively in the shadow of the copy during the integrated csum-copy in Linux-2.4.16; ii) that the checksum operation itself requires minimal cycles. In summary, on new generation processors where execution bandwidth is plenty, checksum computations are not the bottleneck. Instead, it is the movement of data in and out of the processor. Checksumming needs to be done, preferably in conjunction with the data being moved from one buffer to another. Basically, performing checksums when the data is copied from the socket buffers to application space incurs minimal extra costs over hardware checksumming. More importantly, our observations stress the need for reducing copies (rather than checksumming) as the more important optimization.

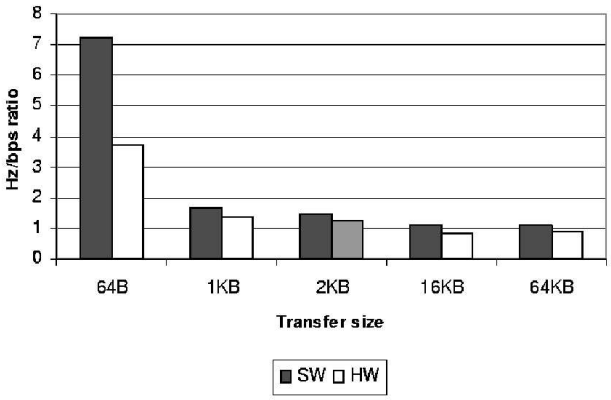

Figure 15. SW versus HW checksum (RX): Hz/bps performance

6 Discussion

Major architectural changes have been proposed to counter known networking bottlenecks [6, 12], but none has been commercially successful and still supports TCP. To reduce kernel overheads, user-level-TCP and other similar OS-bypass methodologies such the Virtual Interface Architecture [12] can be employed, and had done so quite successfully in localized system networks. Among the software "tricks" adopted, the more successful ones include the use of in-kernel applications and zero-copy transmits. In kernel applications (e.g. TUX web server) resides in kernel context, and bypasses latencies introduced by context switches. Their use is limited to a handful of trusted applications closely tied to the OS. Linux-2.4 implemented a

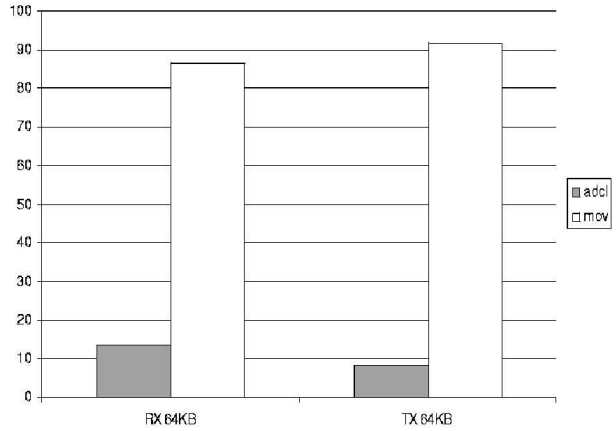

Figure 16. Checksum versus copy (in csum_and_copy routine)

simple version of zero-copy transmits via sendfile(). Sendfile() makes use of the fact that data needed typically already resides in the file buffer cache. The file cache therefore doubles as the socket (kernel) buffer, and data is transmitted directly from these buffers. Without the help of intermediate socket buffers, the application must now carefully monitor page memory usage and release. A page of data must be held as non-usable until the TCP acknowledgements for the associated data arrive. Apache 2.0 now provides for the use of sendfile() wherever the OS supports it [5]. Despite sendfile's limitations, it is one of the easiest and most practical ways to date of achieving zero-copy on transmits. There is no easy solution on the receive side. For now, TOEs seem to be the one solution that addresses more of these problems. Protocol processing is offloaded. TOEs indirectly achieve zero copy, since any data movement is localized in the TOE adapter. Data is ultimately DMAed (without CPU intervention) to the application buffer when the transaction is deemed complete. As such, TOEs also reduce the interrupt processing requirements by interrupting the host processor on a transaction rather than packet granularity. However, many issues remain. TOEs require substantial processing hardware and large amounts of onboard memory. In other words, TOEs simply move the hardware requirements somewhere else. Kernel overheads can still be substantial if the OS's support of TOEs is not implemented well. An ongoing effort in the IETF [3] looks at remote direct data placement and aims to define a standard whereby an intelligent adapter can directly place data in the final application buffers, potentially allowing for true zero copy. To realize the full benefits of zero copy, sockets operations must essentially be asynchronous in nature. The current BSD-sockets interface is closely with the way TCP works. It relies on a copy semantic to/from kernel buffers, and is synchronous in nature (sleeps on blocks). As we have seen in this study, many cycles can potentially be wasted in switching between threads whenever the socket calls are made to wait. In other words, a new generation of sockets programming interface must be adopted [2]. Provisions in operating systems must also be made to support the new sockets paradigm and intelligent offload engines. Applications will have to re-written to make use of these new constructs.

7 Conclusion and Future Work

We have seen that with CPU speeds going beyond 1 GHz, the 1GHz/Gbps rule may not hold. This is especially true of small transfer sizes. The non-processing parts are making a more significant impact on performance as CPU speeds increase. The need for a balanced system is vital. In the mean time, new processor technology, such as hyperthreading that targets fine-grain scheduling [11], will help us hide memory latency at the system level, and lessen that gap. Nevertheless, the reality is that network processing will not scale with CPU speeds. It is doubtful that the incremental offloads seen so far will see us through 10Gbps speeds. The difficulty in offloading parts of TCP processing lies in the fact that there is no one obvious bottleneck across different workloads. Modern NICs coalesce receive interrupts based on some tunable settings. As expected, interrupt coalescing can only be taken so far before response times are affected. Receive checksum offload will get us at best 10% improvement (for large transfers) and segmentation offload [4] is useful again only for large transfers. We have observed a trend that reverses the view that hosts are receive-limited and this has various implications. For a web server, for example, the payload is typically larger on transmits than receives. This observation calls for more attention to be given to optimizing the transmit path in future implementations. Despite all the optimizations that had been implemented, the hotspots of TCP processing remain: copies, interrupt processing, sockets and protocol processing, kernel overheads. Previous studies have not paid much attention to sockets interface, and we have found that this layer makes up a large part of TCP processing. For future work, we plan to investigate the sockets interface. We would also like to study the newer networking models that use event queues and the asynchronous paradigm. Ultimately, a generic asynchronous, zero-copy, TCP-offload paradigm, fully supported by the operating system and exposed to applications, will be required. . The challenge lies in living up all of these expectations, and still maintaining the integrity of the TCP protocol. For now, true low-cost TCP offload remains elusive, with good reason.

Acknowledgements

We would like to thank ShuBin Zhao, Ravi Iyer and Raed Kanjo for their expert input on performance counters and benchmarks. We thank Gary McAlpine and Vinay Awasthi for their pivotal roles in helping us figure out DMA operations and gigabit Ethernet drivers. We also thank Don Cameron and the anonymous reviewers for comments and suggestions on this work.

References

[1] Alacritech SLIC: A Data Path TCP Offload Methodology. *http://www.alacritech.com/html/techreview.html*.

[2] The Open Group: Sockets API extension WG. *http://www.opengroup.org/icsc/sockets*.

[3] Remote Direct Data Placement WG. *http://www.ietf.org/html.charters/rddp-charter.html*.

[4] High Performance Network Adapters and Drivers in Windows. *http://www.microsoft.com/hwdec/tech/*, December 2001.

[5] R. Bloom. Filerting IO in Apache 2.0. *http://www.serverwatch.com/tutorials*, September 2000.

[6] J. Chase, A. Gallatin, and K. Yocum. End-System Optimizations for High-Speed TCP. *IEEE Communications, Special Issue on High-Speed TCP*, June 2000.

[7] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP processing overhead. *IEEE Communications*, June 1989.

[8] A. Earls. TCP Offload Engines Finally Arrive. *Storage Magazine*, March 2002.

[9] G. Herrin. Linux IP Networking: A Guide to the Implementation and Modification of the Linux Protocol Stack. *http://kernelnewbies.org/documents/ipnetworking/*, May 2000.

[10] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *Proceedings of ACM SIGCOMM*, 1993.

[11] W. Magro, P. Peterson, and S. Shah. Hyper-Threading Technology: Impact on Compute-Intensive Workloads. *Intel Technology Journal*, Feb 2002.

[12] G. Regnier and D. Cameron. *The Virtual Interface Architecture*. Intel Press, 2002.

[13] A. Rubini and J. Corbet. *Linux Device Drivers*. O'reilly, 2001.