**RC24245 (W0704-131) April 24, 2007**
**Computer Science**

# IBM Research Report

## Profiling TCP:
## An In-depth analysis of Processing Costs

**Jason LaVoie, Erich Nahum**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

**Robert Flynn**
Polytechnic University
Hawthorne, NY 10532

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

# Profiling TCP: An In-depth Analysis of Processing Costs

Jason LaVoie, Erich Nahum
*IBM T. J. Watson Research Center*
*Hawthorne, NY, 10532*
{lavoie,nahum}@us.ibm.com

Robert Flynn
*Polytechnic University*
*Hawthorne, NY, 10532*
flynn@poly.edu

## Abstract

There are currently several proposals for large, disruptive changes to the Linux TCP implementation to improve network performance and scalability. These proposals, including relocating part or all of the TCP stack, header splitting, and dedicated network caches, require potentially sizeable changes to the Linux TCP implementation. Before considering these modifications, we believe the behavior of the Linux TCP stack should be revisited to be better understood, studied, and optimized.

We provide an in-depth analysis of CPU profiles for both the Linux 2.4 and the 2.6 TCP stacks. For Linux 2.4.16, we vary several aspects of networking and profile the TCP stack for a detailed look at how the code path changes under load. We show that after years of development and enhancements the limitations of high performance networking remain the same even across different kernel versions and hardware.

For the Linux 2.6.14 TCP stack, CPU profiling is performed using callgraph support from OProfile. The callgraphs and newly developed callgraph examination tools are used to determine where high level TCP functions, such as TCP receive and TCP send message, are spending CPU cycles. These results are then examined to determine bottlenecks and potential areas for improvement with respect to CPU usage. Under load, we show why TCP receive dominates costs even for large files.

For each kernel, we propose several possible non-disruptive enhancements. Furthermore, we present correlations and trends from the 2.4 to the 2.6 TCP stack and trends over generations of machine hardware. Our results validate and expand upon existing TCP profiling work.

## 1 Introduction

Research has shown that on average TCP processing alone consumes about 30% of the CPU on high volume HTTP servers [8], [3]. This large cost of TCP processing on high volume network servers has been the focus of much recent research. Much of that work is interesting but requires large sweeping changes to TCP and/or changes to the socket API. Several proposals exist for TCP relocation, such as TCP Offload Engines (TOEs), on-loading, and user-space TCP. Other slightly smaller, but still intrusive, enhancements include moving header splitting and acknowledgement coalescing to the Network Interface Card (NIC). While this research is certainly interesting and useful, the motivation with respect to TCP's behavior has been ignored. The Linux TCP implementation is a complex set of functions whose cycle cost and interaction could be better appreciated by the larger community. Data profiling the Linux TCP implementation is scarce and dated. Before making large sweeping changes to the Linux TCP stack, these functions and their interactions should be better understood. Many cite the costs of the TCP as their motivation; however, they lack the behavioral studies that are key to supporting the vast amount of code changes.

Most studies covering the behavior of TCP are either dated or look at the data at too high of a level. Others lack details to thoroughly understand what is happening in the system. Our contributions include revalidating previous assertions regarding the behavior of TCP and enhancing the understanding of the known limitations to TCP using advanced techniques.

Using kernel profiling and several specially developed tools, a novel approach to TCP behavioral profiling is detailed and performed herein against the Linux 2.6 TCP implementation. The results are then compared against profiles of the 2.4 Linux TCP implementation. While the Linux 2.4 TCP stack may seem uninteresting for determining the future, comparing the different generations of the kernel is helpful in identifying trends. We present a detailed inspection of TCP to motivate the appropriate areas to examine with respect to TCP performance. Using new tools, we demonstrate that TCP receive dominates CPU cycles in send heavy workloads even for large files and provide an explanation as to why. We also use our analysis to highlight possible areas for improvement.

The motivation for this analysis is the cost of TCP processing on high performance network servers, and the trends indicating it will worsen with faster wire pro-

tocols. While the cost is largely understood, the reasons are not because studies into the behavior of TCP lack details and depth. We study the behavior of TCP cycle consumption across Linux versions, hardware, HTTP servers, clients, and workloads. Linux was chosen not because it lacks decent performance. On the contrary, Linux was chosen because it contains the latest enhancements and is the focus of the most recent research. This is an analysis of the implementation, not of the TCP protocol itself. Our results validate previous studies [7], [8] of TCP while providing more detail and insight on newer hardware using the latest TCP enhancements.

## 1.1 Paper Outline

The rest of this paper proceeds as follows. Section 2 presents an analysis of the Linux 2.4.16 TCP stack. Section 3 presents a detailed look at the 2.6.14 TCP stack. Section 4 identifies trends and similarities between the two, Section 5 covers related work, and Section 6 summarizes our contributions and plans for future work.

# 2 2.4 Profile

To identify and solve performance bottlenecks in the TCP/IP stack via measurement and profiling, the Linux 2.4 stack was examined. An HTTP server workload was chosen because much work has already been done in this area for comparison purposes. In addition, multiple servers and workload generators were available for a breadth of experiments. To show trends in TCP behavior from 2.4 and 2.6, we ran experiments using older hardware that would have been appropriate for the time when Linux 2.4 was recent.

The server under test was an IBM Intellistation M-Pro, 400 MHz Pentium II with 320 MB of RAM and an Acenic gigabit adaptor, running Linux 2.4.16 UP. Eight clients were used to generate HTTP requests. These were 500 MHz Pentium III's with 100 Base-T adaptors running FreeBSD 3.3. The server was connected to an Alteon ACESwitch 180 and the clients to a Nortel BayStack with a gigabit uplink to the Alteon ACESwitch.

To test the TCP stack under various server software, several HTTP servers were used to serve content: Apache 1.3.20, Apache 2.0 [21], the Tux in-kernel HTTP server [11], and the Flash HTTP server [20]. In addition, several different I/O event notification methods were employed. These methods are: select, poll, /dev/epoll, and real-time signals with sig-per-fd. The Flash web server was used for all those experiments. The /dev/epoll code is from Davide Libenzi's /dev/epoll implementation on 2.4.16 [14]. The signal-per-fd code is based on an idea from Chandra and Mosberger [5], and the code was taken from a patch by Vitaly Luban [15].

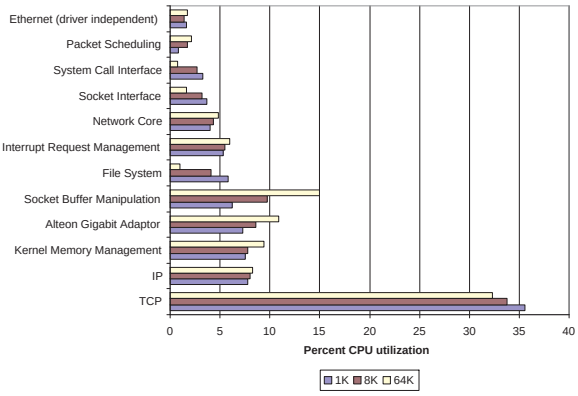


Figure 1: 2.4.16 CPU util. breakdown by category for different file sizes

## 2.1 Tools

Three workload generators were used in these experiments: WaspClient, SingleClient, and the SPECweb99 client. WaspClient [18] is designed to produce a realistic traffic mix, generating a range of HTTP 1.0 requests that capture characteristics of web server workloads. Examples of these characteristics include log-normal file size distributions, Zipf popularity, and embedded object requests. In this benchmark, load is defined in user-equivalents with higher numbers of user-equivalents producing more load.

SingleClient [25] is used as a stress-testing tool to produce HTTP 1.0 requests for the same file. The user can increase the load by raising the number of simultaneous concurrent connections. Our experiments with SingleClient increase the load until the system under test is saturated.

SPECweb99 [24] is an industry standard benchmark used to gauge several aspects of HTTP servers. In these experiments, dynamic content was turned off, and the working set size was reduced such that it would fit into RAM.

CPU utilization profiles were obtained using the version of OProfile [19] that was included with Linux 2.4.16. All results are based on the CPU being 100% utilized serving HTTP responses.

## 2.2 Results

The SingleClient HTTP traffic generator was used to request different file sizes from the Flash HTTP server. Figure 1 shows the percentage of total system CPU usage by general category, without locking, for each requested file size. As seen by Foong et al. [8], TCP code typically takes 30 percent of the total CPU cycles in the experiments, followed by the socket buffer (skb or sk buffer) and Acenic routines at roughly ten per-

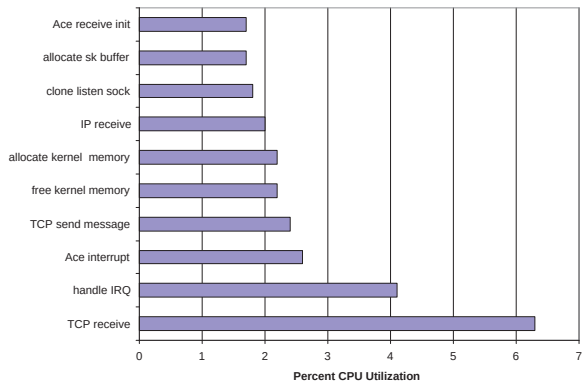


Figure 2: 2.4.16 top functions for 1K file transfers

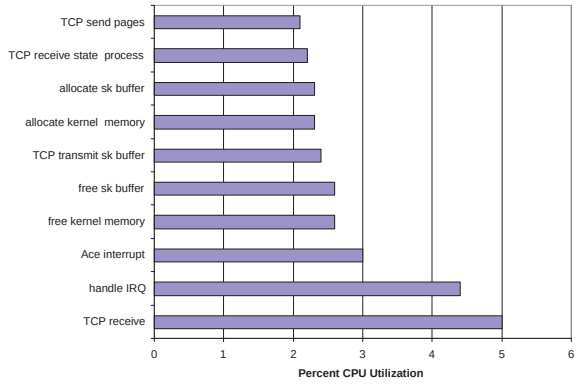


Figure 3: 2.4.16 top functions for 8K file transfers

cent each. These results also match that of Foong et al. [8] and validates previous work showing the high cost of the operating system, specifically memory management and socket buffer manipulation. Very similar results were found for Apache 1.3, Apache 2.0 and Tux using WASPclient, SPECweb99, and with other configurations of SingleClient.

We then looked at the top CPU consuming functions per profile and graphed them per requested file size. Figures 2 - 4 show the top cycle consuming functions and their cycle consumption for 1k, 8k, and 64k files. Each of these graphs are dominated by TCP send and receive and kernel memory management functions. Functions prefixed with "Ace" belong to the Acenic adaptor. More fine-grained profiling data was collected but left out for brevity. As with the categories, the top functions remained fairly consistent over the varying workloads and servers.

The following is a list of the possible candidates for Linux 2.4 TCP stack optimizations based upon the web server workload and kernel profiling results observed:

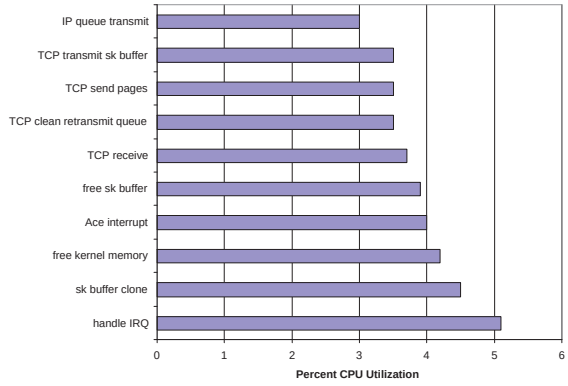


Figure 4: 2.4.16 top functions for 64K file transfers

- *TCP receive* takes up to five percent of the CPU cycles. This function is found in virtually all the profiles, and it is responsible for the initial demultiplexing of arriving TCP segments.
- *TCP send pages* takes three percent of cycles. This function allocates new buffers for messages for the TCP send side, does a virtual copy, and queues the messages into the send queue for the subsequent call to TCP write transmit.
- *TCP transmit sk buffer* takes about 2.8 percent of the CPU cycles. This function builds the TCP headers for sk buffs about to be sent and then calls the network layer queueing routine. The headers are explicitly built each time rather than doing something more efficient such as copying a header prototype and then only modifying the relevant fields as BSD does.
- *sk buffer clone* takes about 2.3 percent of cycles and performs a virtual copy of an sk buffer. It explicitly copies field-by-field rather than doing something more efficient like a structure assignment.
- *TCP clean retransmit queue* typically takes two percent of cycles. It cleans out packets in the retransmission queue once they have been acknowledged from the other side. Currently, it looks at all segments in the send queue rather than breaking out of the loop once all relevant bytes have been cleaned.
- *TCP receive state process* takes two percent of cycles. This function implements the receiving side processing for a connection (after it has been identified) for all states except ESTABLISHED and TIME_WAIT, i.e., for connection setup and teardown.
- *TCP clone sock* typically uses one percent of cycles. This function creates a new socket from the listen socket. Currently, it explicitly zeros out fields

in the socket data structure rather than using structure assignment (as BSD does) or through a memset or bzero function.
- *handle IRQ*, which manages interrupt processing, typically consumes two to five percent of cycles. This improved with Igno Molnar's IRQ patch; however, there still appears to be room for improvement.

## 3 2.6 Profile

After examining the Linux 2.4 TCP stack under load, we revisited the experiments using a newer kernel, newer hardware, and improved tools. The server under test was an IBM XSeries 345, 3.2 GHz Pentium Xeon with one GB of RAM, four Intel E1000 gigabit adaptors, running Linux 2.6.14.2 SMP without hyperthreading with OProfile enabled and CONFIG_FRAMEPOINTER enabled. An SMP kernel was used so the effects of locking could be observed. Thirteen clients were used to generate HTTP requests. These were 1.7 GHz Pentium 4's with Intel E1000 gigabit adaptors running Linux 2.6.5. All of these machines were connected to a pair of linked Dell gigabit switches. Clients were configured to connect to a single card on the server where three of the server's cards had three clients and one had four clients. In every experiment, the server's CPU was 100% utilized serving HTTP content.

To allow some comparisons to be made with the 2.4 results, the client generator, SingleClient [25] was used as well as the Flash HTTP server [20]. Those experiments were conducted with only eight of the clients. A set of persistent connections experiments were conducted using HTTPerf [17] and Autobench [16] with the number of requests per connection set to ten and using all 13 clients.

### 3.1 Tools

#### 3.1.1 Profiling

A profile callgraph provides the key information required to determine the callee-caller relationships as workload parameters are varied. A profile callgraph can also dive deeper more easily than a static analysis of source code. Using a callgraph, sub functions at an arbitrary depth can be located and the cost to the higher level function can be assigned. For example, a function foo taking 10% of the CPU cycles may have a called function that is m call levels deep requiring 3% of the cycles. This called function may not show up in the list of top cycle consumers and given the depth of m, may not be known to contribute to the cost of foo.

OProfile [19] provides a kernel profile of functions sorted by the number of CPU cycles used per function. When the callgraph option is used, each function listed is accompanied by a list of callers and callees. The callers are determined by examining the call stack when a sample is taken. A percentage per caller is given, and it represents the relative percentage of other callers for the respective function looked at. In other words, for a given function, when a sample is taken, a percentage is increased for functions seen on the stack relative to other functions.

All profiling experiments used the following options. OPControl was run with *separate* set to kernel, *vmlinux* set to the local executing 2.6.14-2 Linux kernel, and *callgraph* set to five. Varying the callgraph depth did not alter the results. OPReport was executed with the callgraph option and the -l option set to the local executing 2.6.14-2 Linux kernel.

#### 3.1.2 Callgraph Parsing

To extract needed information from the large callgraph output from OProfile, a special tool was developed. This tool, Greatest n Descendants, shows the greatest n descendants for the top m cycle consuming functions. The descendants can be arbitrarily deep in the callgraph rooted at one of the top m functions. A top function is one listed in the Oprofile output as taking the largest percentage of CPU cycles during the profiling experiment.

Given a callgraph generated via OPReport as input, and the parameters m and n, Greatest n Descendants first creates m call trees rooted at the mth greatest function. These trees are created in the same physical data structure - meaning identical functions per graph share a node in this structure.

After the call tree is assembled, it is traversed via a Breadth First Search (BFS) per top node. While traversing the tree, n descendants with the highest CPU usage percentage are collected. The result is a list of the n greatest descendants of each tree. Having each descendant listed with cycles consumed is quite insightful; however, it does not provide enough information. If two higher order functions use the services of a lower level function, only a certain percentage of the cycles consumed by the lower level function can be attributed to each higher order function to properly map the behavior.

#### 3.1.3 Attribution

Greatest n Descendants gives us the top cycle consuming functions for the high level functions; however, it is not possible to tell how much of the descendant's costs belongs to a top function (i.e. the root of a call tree). Using the Linux source tree, the callgraphs from OProfile, and the results from the Greatest n Descendants, graphs for each set of results were made from the interesting top functions down to and past their Greatest Descendants. Generic versions of these graphs can be found in figure 5 for TCP receive, figure 6 for TCP's sending routines, and

figure 7 for sk buffer allocation. Italicized functions in these diagrams denote functions that appear in more than one diagrams (i.e. they cross diagrams). For simplicity, the figures only include the relevant functions that appear in our profiles as well as some additional functions that give context. All of the results presented in this section correlate to these figures, thus one can see the potential call tree for a given function with respect to the TCP operation. The edges per graph and weight of the edges varies with different workloads.

The generic callgraph rooted at *TCP receive* can be seen in figure 5. *TCP receive* is animated by the IP layer (specifically *IP local deliver*) when a new packet arrives off the network. A typical processing path for an inbound packet is to *TCP receive established* through *TCP do receive*. If the packet contains an acknowledgement, *TCP ACK receive* is called, and then a new acknowledgement is created in *TCP send ACK* (via *TCP ACK send check*). A new sk buffer is allocated and then sent via *TCP transmit sk buffer*. The path from there can be seen in figure 6.

Figure 6 has two primary roots, *TCP send message* and *TCP send pages*. *TCP send message* is the standard send mechanism for TCP via the send socket call. *TCP send pages* hands scatter-gather page lists to the Network Interface Card (NIC). It is used with the sendfile socket call. A standard send will see *TCP write transmit* called for the pending sk buffer. This in turn calls *TCP transmit sk buffer* which calls *IP queue transmit*. The call tree can be followed all the way down to the device including the device queueing mechanism. A secondary root to the graph in figure 6 is *TCP receive* (connected via *TCP receive state process*). This relationship is explored in more detail later in this section.

The allocation of sk buffers, seen in figure 7, is based on Bonwick's Slab Allocator [4]. The basic idea is to keep some amount of free memory available for allocations. If that amount of memory drops below some low water mark then more is allocated. Some functions in this graph are also animated by the poll system call.

A new tool, Calltree, was developed to produce reduced callgraphs or sub-callgraphs. This tools allowed us to create reduced callgraphs with only those functions that were interesting while trimming out the noise from the larger generic callgraphs presented above. Given a list of functions as nodes, callers with a calling percentage greater than a threshold are traversed. Thus a call tree with the greatest callers can be built bottom-up. With this information, the cumulative attribution can be calculated creating attribution sub-callgraphs. These show the cost in cycles per graph node for the higher level function of interest. This allows us to attribute the cost of a function to its ancestors.

## 3.2 Results

Figures 8 - 10 show the top ten functions for CPU utilization and their respective CPU utilization as reported by OProfile for various requested file sizes. As the file size increases, so does the cost of sending pages (Flash uses the sendfile socket call). For smaller files, the costs of locking and memory management is very high. In addition, as file sizes increase, the relative cost (in CPU utilization) of sending increases (TCP send message is 1.25% for 1K files) while the relative cost of receiving decreases. As with the 2.4 kernel, *sk buffer clone* has an increasing cost as the transferred file size increases. This is because less time is spent on connection set-up, tear-down, and look-up, and more time is spent sending data. Even though sending is the primary operation operation for the HTTP server, even with a large file size, *TCP receive* dominates TCP costs. This cost; however, is reduced as the file sizes increase.

Transmit Segmentation Offload (TSO) is a feature offered by many NIC vendors that advertises a higher network MTU (typically 64K) to TCP . TCP will then give send buffers to the NIC in these larger units. The NIC then breaks these larger buffers into packets of the actual MTU size. This leads to less memory and cycles consumed by TCP because it is dealing with larger buffers. It also should reduce the amount of time spent communicating with the NIC. Figures 11 - 13 show the results of the same experiments as before but with TSO enabled. For 8K files, *TCP send pages*, the routine used to hand pages to the NIC when using TSO, is ranked 11th with a CPU utilization of 1.6%. For larger files, the increase cost of send is less pronounced with TSO enabled and receive is more pronounced. The cost of cloning an sk buffer also increases with file size. With and without TSO, for small files, the system call layer hook is a top ten function. A hook for the system call layer is executed whenever system call is executed from user space.

Larger files also had file reference count functions in the top ten functions when persistent connections were used. This is expected because less cycles will be consumed with connection setup and tear-down. Persistent connections also showed greater cycles consumed by TCP poll (approximately 4%) because connections are kept in the system longer, therefore, poll has more file descriptors to check.

The callgraphs and the cumulative sub-callgraphs allowed us to determine the drag of other functions attributed to top level TCP functions. Not only do we look at the cost of a function say, *TCP receive*, we drill down and look at those functions that are called as a result thereof and attribute the cost. We used the Greatest n Descendants tool to pull out the highest descendants for *TCP receive*, *TCP send message*, and *TCP send pages* for each of the file sizes. However, as seen in figures 5

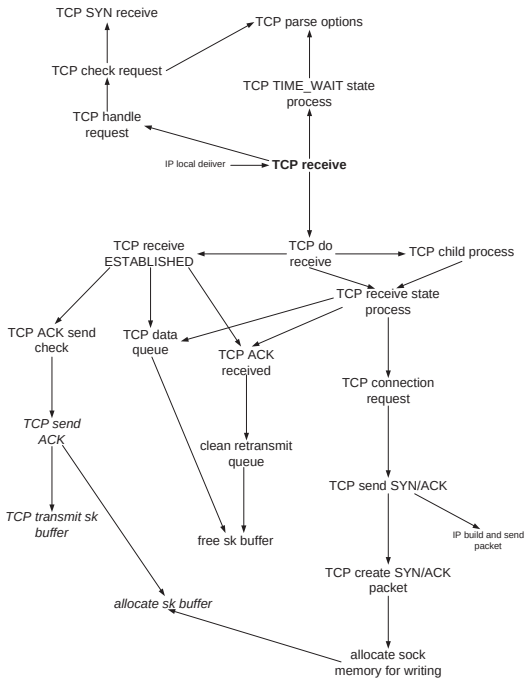


Figure 5: Generic callgraph for TCP receive

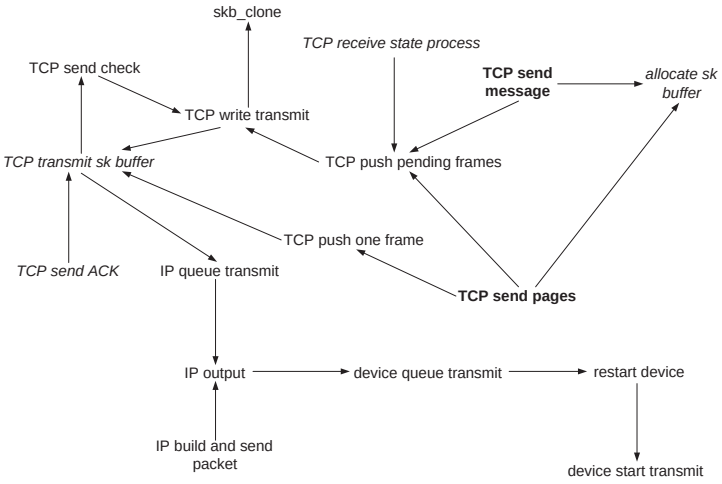


Figure 6: Generic callgraph for TCP send message and send pages

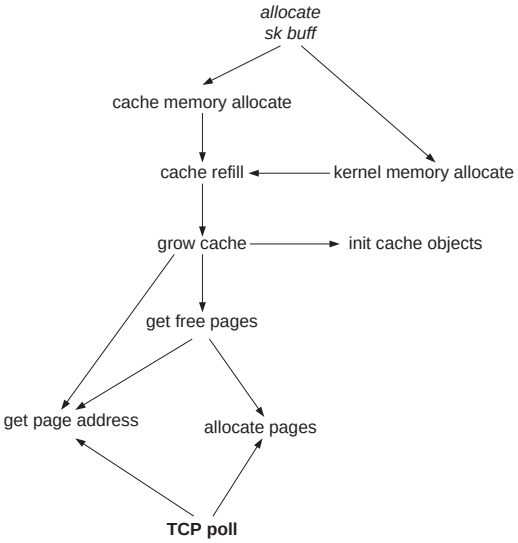


Figure 7: Generic callgraph for sk buffer allocations

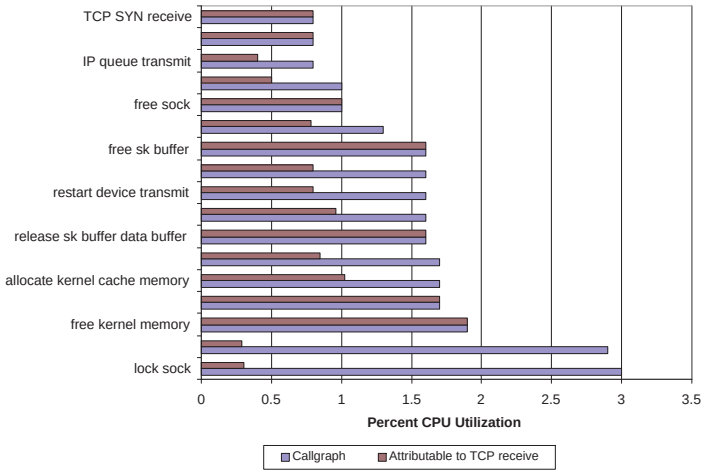


Figure 14: TCP receive sub-function utilization with attribution for 1k files

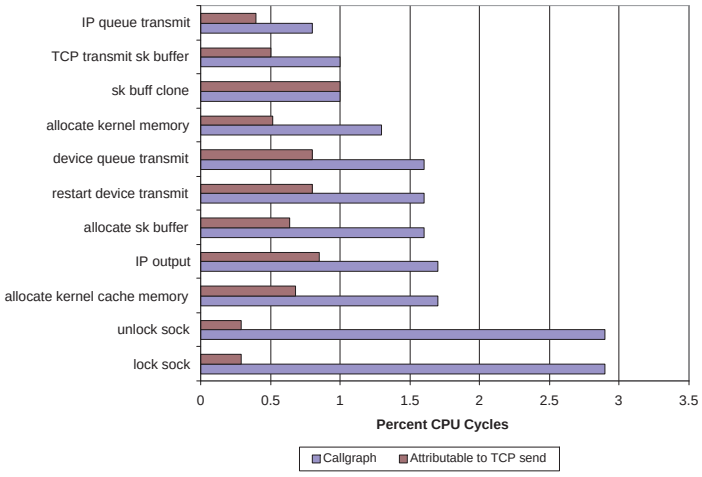


Figure 15: TCP send message sub-function utilization with attribution for 1k files

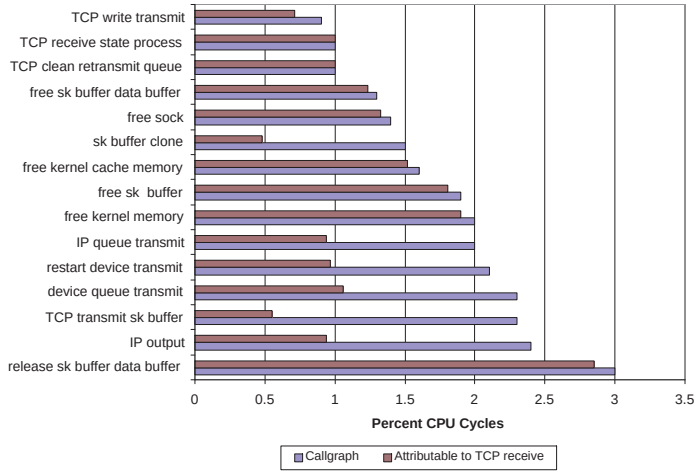


Figure 16: TCP receive sub-function utilization with attribution for 8k files

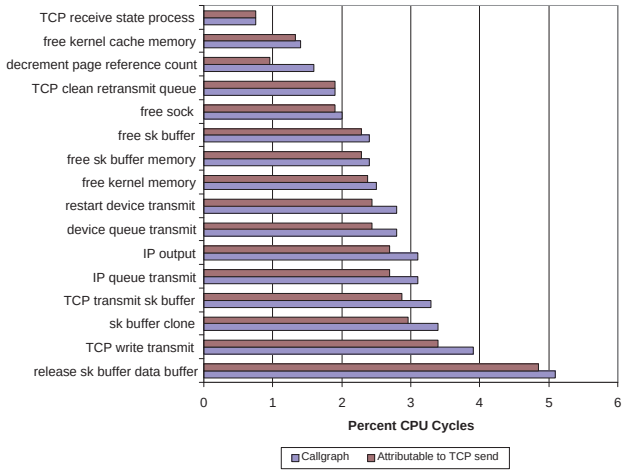


Figure 17: TCP send pages sub-function utilization with attribution for 8k files

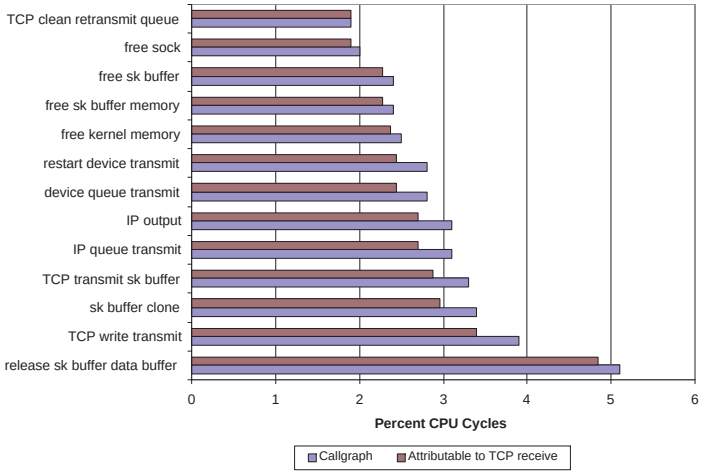


Figure 18: TCP receive sub-function utilization with attribution for 64k files

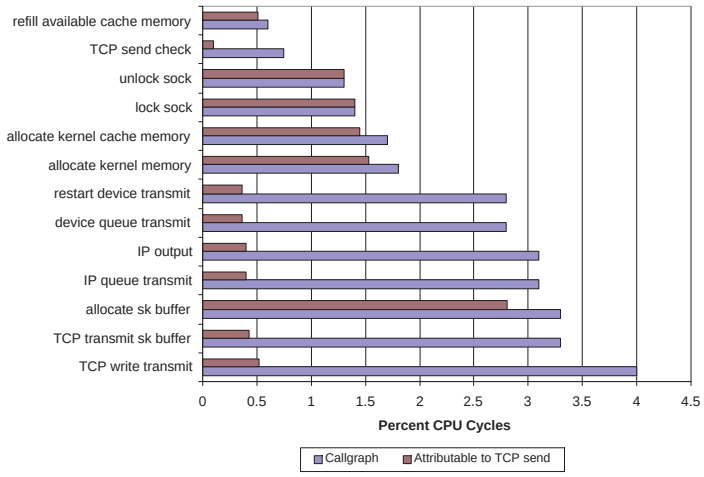


Figure 19: TCP send pages sub-function utilization with attribution for 64k files

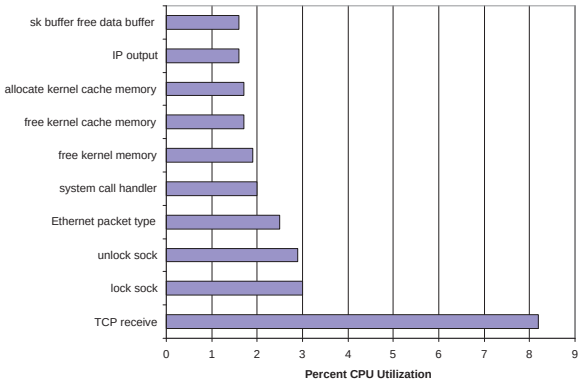


Figure 8: 2.6 top functions for 1K file transfers

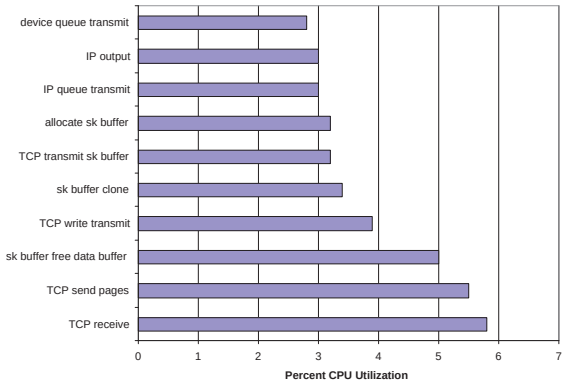


Figure 10: 2.6 top functions for 64K file transfers

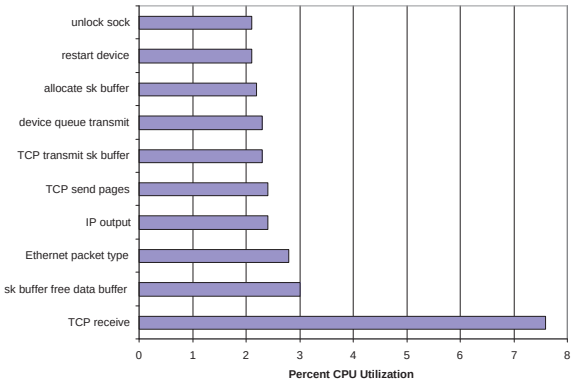


Figure 9: 2.6 top functions for 8K file transfers

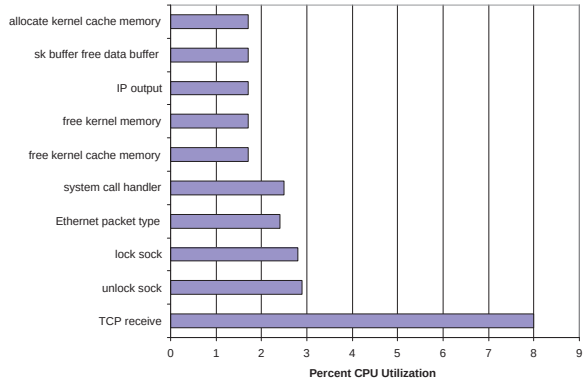


Figure 11: 2.6 top functions for 1K file transfers with TSO

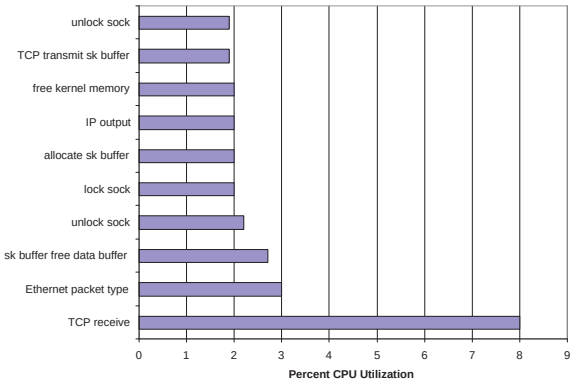


Figure 12: 2.6 top functions for 8K file transfers with TSO

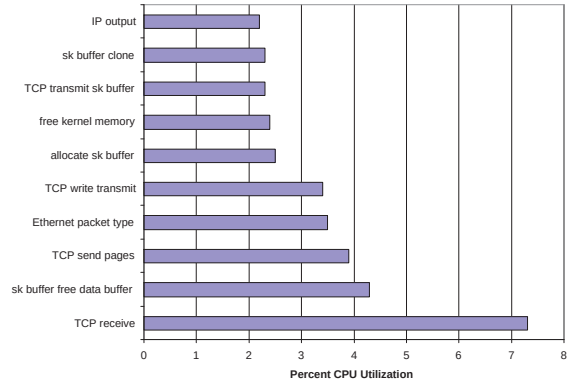


Figure 13: 2.6 top functions for 64K file transfers with TSO

- 7, multiple paths may exist to a given function. Thus, we also calculated the attribution of a called function to each major TCP operation. Figures 14 - 19 show the called functions by high-level caller with the percent of CPU cycles consumed and the approximate percentage of cycles consumed by the callee that can be attributed to the caller. For example, in figure 14, ten percent of the three percent of cycles used by *lock sock* can be attributed to *TCP receive*. Identical TSO experiments were conducted and the percentages observed were very similar.

*TCP send message* becomes less significant as the file size increases; hence, it is excluded from figure 17. In the 8K transfer experiments, *TCP send message* is attributed with only 16% of sk buffer clone, 21% of *IP output* and 14% percent of sk buffer allocations. As file sizes increase, *TCP receive* becomes responsible for those costs. This is because incoming packets can open the window size and can send queued packets. This behavior can be seen in figure 6 where *TCP receive established* calls *TCP push pending frames*. For 64K files, *TCP receive* was responsible for 96% CPU time spent in *TCP push pending frames*. This indicates a large amount of queueing for sends.

Using just the high level profiles, locking appears to be an issue. However, we see in the attribution graphs 14 - 19 locking is distributed. For smaller file sizes, locking has a higher concentration in connection setup and look-up. When persistent connections are used, as the file size increases, the locking primarily falls under *TCP send message*, and it becomes a larger percentage of total machine cycles. Freeing sk buffers always shows up as a cost attributed to *TCP receive* because it processes the ACK packets that cause sk buffers to be freed. Processing acknowledgements, specifically cleaning out the retransmit list and sending pending packets, has a large cost regardless of file size.

Figure 15 shows that *TCP Send* is really only responsible for about 50% of the one percent of cycles used by *TCP transmit sk buffer* for 1K files. The cost of sending an acknowledgment makes up most of the difference with 41% of the cost of *TCP transmit sk buffer*. In addition, 41% of cost of every function called below *TCP transmit sk buffer* (*IP output*, *device queue transmit*, etc.) is attributable to sending an acknowledgment. Similarly, figure 17 shows that sending acknowledgments is attributed with 23% of the sending costs in the 8K case. Therefore, offloading ACK generation may be beneficial. The largest cost we see is in the memory management. Frequent allocation and deallocation of sk buffers is costly across all file sizes. We plan to analyze this further by dissecting the code and analyzing the cache behavior. The aggravation in the latter may be increasing the cycle counts for the memory management func-

tions. The costs of memory management is unexpected because the Linux slab allocator has been optimized.

Across all of the attribution graphs, we see *TCP receive* carries most of the processing costs. Across the file sizes, kernel memory deallocation is attributed to *TCP receive*. Also, approximately half of kernel memory allocation is attributed to *TCP receive*. Figure 19 shows the cost of allocations and locking moving towards *TCP send pages* as file sizes increase. Figure 16 and 18 show the costs of sending at the device to be more attributable to *TCP receive* than *TCP send pages*. This is likely due to the efficient sending of large buffers from *TCP send pages* versus the sending of smaller buffers (acknowledgment packets with no application data) from *TCP receive*.

## 4 Comparison between Linux 2.4 and 2.6

There are many similarities between the profiles seen for the Linux 2.4 and 2.6 kernels. Several functions, e.g. *TCP send pages*, even show the same percentage of cycles used. There are some differences; however, that are likely explained by the different ethernet NICs. IRQ handling was more prevalent in the 2.4 profiles. This is likely because network interrupt mitigation via New API (NAPI) [23] has reduced the number of times the operating system is interrupted. This is done by converting from interrupt mode to polling mode after an interrupt has arrived. Once there are no more incoming packets to process, interrupts are turned back on. This can be confirmed with more experiments by turning off NAPI. We also see an increase in relative receive processing in the 2.6 TCP implementation likely because of NAPI.

Graphs 20 - 22 display the similarity for exact functions that showed up in both kernel's top ten functions. Some similar functions are not shown. As the served file size increased, the number of identical functions between the two kernels' top ten converged. The cost of locking is not seen in the 2.4 kernel because a uniprocessor kernel was used.

For all three file sizes, the cost of TCP receive has increased for the 2.6 TCP stack. Given the attributions seen in the previous section, the trend is likely to continue. For the larger file size, the cost of *TCP send pages* in 2.6 grew over 2.4 likely due to enhancements in other areas of the Linux kernel.

Overall, the profiles are very similar, and for some functions, identical. As the transferred file size increased, the similarity converged. Although the interaction with the Linux kernel memory system has changed, the costs remain high. Given all of the work that has gone into Linux between 2.4 and 2.6, the similarities in the profiles is surprising.

Another interesting trend is seen in the hardware. Although the capabilities of the hardware increased be-

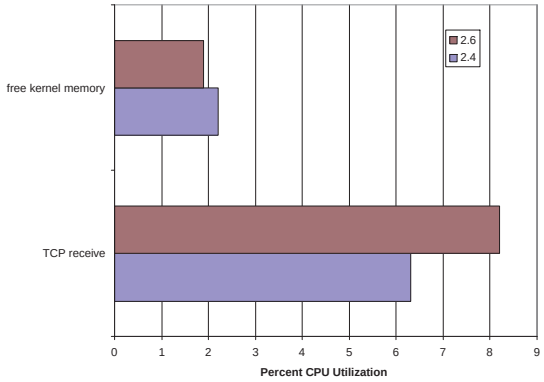


Figure 20: Kernel comparison for 1K file transfers

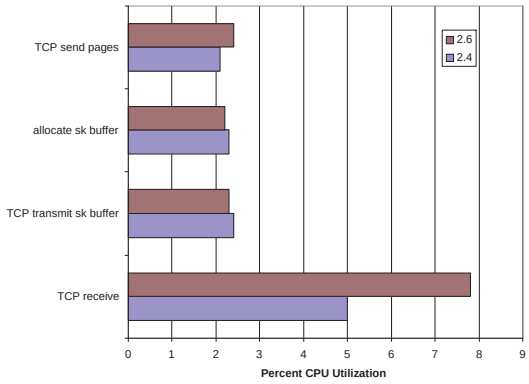


Figure 21: Kernel comparison for 8K file transfers

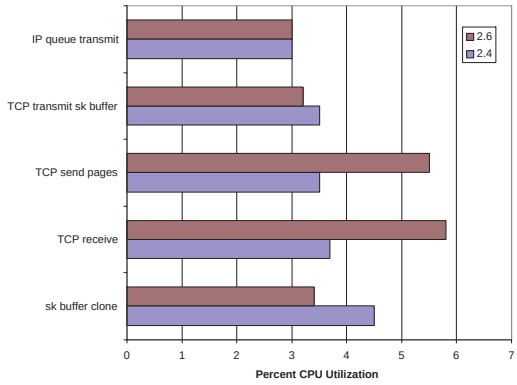


Figure 22: Kernel comparison for 64K file transfers

tween the 2.4 profiles and the 2.6 profiles, the behavior of the TCP stack remains similar. This helps support the theory in [9] that TCP is not scaling with host processor speeds.

## 5 Related Work

Early work by Clark et al. show the implementation of TCP, not the protocol itself, as the bottleneck [7] in high performance network processing. Operating system overheads, e.g. memory, timers, etc. dominate the network processing costs. Many proposed enhancements from [7] have been incorporated into TCP.

Many years ago, Kay and Pasquale instrumented a DEC Ultrix 4.2a kernel and showed the cost of non-data touching operations remaining roughly constant over various message sizes [12]. This work predates checksum offload; therefore; data touching operations and movement were dominant for increasing file sizes. They also highlighted the need for optimizations to non-data touching operations to produce a significant performance improvement. Similar work [13] by the same authors helped motivate the addition of checksum offload and concludes a wide range of small improvements are needed for non-data touching operations to improve TCP latency.

Bhattacharya and Apte compare Linux 2.4 and 2.6 TCP implementations [2]. Time spent in the socket calls socket and bind is greater in 2.6 versus 2.4, while listen and connect time is less. They determine that connection setup and tear-down in 2.6 requires more processing than in 2.4. Dramatically better HTTP throughput using Apache [21] and HTTPerf [17] as well as a lower response time was achieved with 2.6 as load/connections increased. A breakdown of TCP processing overheads for a single connection shows the NIC driver code, interrupt processing, buffer copying, and checksumming as the most CPU intensive operations during TCP packet processing. With a Linux 2.6 kernel, it is not clear why checksum offload was not used.

Anand and Hartner [1] presented a detailed look into the Linux 2.4 and 2.5 TCP stacks with a focus on TCP scalability for SMPs and Gigabit networks. Hardware issues prevented a complete and thorough investigation. Some callgraph profiles were presented with a primary focus on interrupts.

Foong et al. [8] evaluated Linux 2.4.16 and Windows 2000 using high-level categories for small data transfers. Kernel overheads, sockets, and protocol processing were shown to take up the majority of cycles. Data touching and interrupts represent the bulk of cycles spent for large transfers.

Kim et al. [26] evaluate TCP at a system level by examining cache misses and instruction counts. They examine the effects of the system when moving the placement of the NIC; however, they do not probe deeply into the behavior of TCP.

Several others have profiled TCP [3, 22, 6, 10] to varying degrees - usually based on categories. None of the past research has gone to the depth which we are looking. None of the past research have traced callgraphs nor attributed the cost.

## 6 Summary and Future Work

In this paper, two Linux kernel profiles were analyzed with respect to TCP networking. Using OProfile to trace CPU cycle consumption, several possible enhancement suggestions were made regarding the kernel. In addition, light has been shed on the Linux 2.6 TCP stack behavior using callgraph analysis. TCP's behavior and costs have remained high even through a generation of code enhancements, and the bulk of the cost falls into interactions with the operating system: mainly the memory system and interrupts. Targeted enhancements based on our analysis will likely lead to gains in TCP performance.

The function to determine the Ethernet packet type consumed between one and three percent of machine cycles in all of the experiments. This function is called by the E1000 NIC to determine an incoming packet's protocol identifier. Since checksum offload is being used, the adapter already assumes the protocol; therefore, this function seems wasteful. This can be easily offloaded to the NIC.

Socket buffer cloning is done on outbound packets to allow packets to be sent along multiple routes. The implementation of this function can easily be shrunk to half the code size using structure assignment. While large gains are not likely, given the frequency of calls to clone an sk buffer, some gains are possible.

TCP receive is responsible for more costs from Linux 2.4 to 2.6. Since most high volume network servers focus on sending, this trend will likely increase. Our attribution graphs showed that sending acknowledgment packets (as part of TCP receive) is responsible for up to 41% of sending costs. Furthermore, for each ACK sent, an sk buffer must be allocated, placed on the retransmit queue, and deallocated. All of these functions also have significants costs that can be reduced by up to 41%. While some propose to offload all of ACK generation to the NIC, this will likely alter all of TCP's timer and round trip calculations. Instead of such a drastic change, we suggest moving the *TCP send ACK* function to the NIC for large gains. This is much simpler and requires fewer changes to TCP.

Based on the data presented herein, we plan to take a closer look these proposed enhancements and determine the potential gains for each. More detailed experiments and analysis thereof may allow us to propose other, more beneficial, augmentations to the Linux TCP implemen-

tation. Finally, we would like to analyze the cache behavior of TCP to determine why memory management of sk buffers is so costly.

Overall, we conclude the behavior of TCP is largely invariant between workloads, HTTP servers, and hardware. Although many enhancements have been added to the TCP implementation from 2.4 to 2.6, the bottlenecks remain the same as those seen by Clark et al. [7]. Enhancements such as TSO have lessened the effects of the bottlenecks but do not eliminate them. More targeted solutions based upon TCP's behavior are proposed to address the TCP bottleneck.

## References


[1] Vaijayanthimala Anand and Bill Hartner. TCPIP Network Stack Performance in Linux Kernel 2.4 and 2.5. In *Proceedings of the Linux Kernel Symposium*, Ottawa, Canada, June 2002.

[2] Shourya P. Bhattacharya and Varsha Apte. A Measurement Study of the Linux TCP/IP Stack Performance and Scalability on SMP systems. In *In the Proceedings of the 1st International Conference on COMmunication Systems softWAre and middlewaRE (COMSWARE)*, New Delhi, India, Jan 2006.

[3] Nathan L. Binkert, Lisa R. Hsu, Ali G. Saidi, Ronald G. Dreslinkski, Andrew L. Schultz, and Steven K. Reinhardt. Performance Analysis of System Overheads in TCP/IP Workloads. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, Sept 2005.

[4] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *In Proceedings of the USENIX Summer Technical Conference*, Boston, June 1994.

[5] Abhishek Chandra and David Mosberger. Scalability of Linux Event-Dispatch Mechanisms. Technical report, Internet and Mobile Systems Laboratory, HP Labs, December 2000.

[6] J. Chase, A. Gallatin, and K. Yocum. End system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

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

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

[9] Douglas Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey. Server scalability and TCP offload. In *USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[10] Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[11] Red Hat Inc. The Tux WWW server. `http://people.redhat.com/~mingo/TUX-patches/`.

[12] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *SIGCOMM*, pages 259–268, 1993.

[13] Jonathan Kay and Joseph Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4(6):817–828, December 1996.

[14] Davide Libenzi. Improving (network) I/O performance. `http://www.xmailserver.org/linux-patches/nio-improve.html`.

[15] Vitaly Luban. signal-per-fd linux kernel patch. `http://www.luban.org/GPL/gpl.html`.

[16] Julian T J Midgley. Autobench. `http://www.xenoclast.org/autobench/`.

[17] David Mosberger and Tai Jin. httperf – A Tool for Measuring Web Server Performance. In *Proceedings 1998 Workshop on Internet Server Performance (WISP)*, Madison, WI, June 1998.

[18] Erich Nahum, Marcel Rosu, Srini Seshan, and Jussara Almeida. Wide-area server performance (wasp) project. In *Proceedings ACM Sigmetrics*, Cambridge, MA, June 2001.

[19] OProfile. OProfile - A System Profiler for Linux. `http://oprofile.sourceforge.net/`.

[20] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[21] The Apache Project. The Apache WWW server. `http://httpd.apache.org`.

[22] Greg Regnier, Srihari Makimeni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP Onloading for Data Center Servers. *Computer*, 37(11), November 2004.

[23] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *Proceedings of the*

*5th Annual Linux Showcase and Conference*, Oakland, Nov 2005.

[24] SPEC. SPECweb99. `http://www.spec.org/osg/web99/`.

[25] Min Tian, Thiemo Voigt, Tomasz Naumowicz, Hartmut Ritter, and Jochen Schiller. Performance impact of web services on internet servers. In *IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, Marina Del Rey, USA, Nov 2003.

[26] Hyong youb Kim and Scott Rixner. TCP Offload through Connection Handoff. In *Proceedings of EuroSys 2006*, Lueven, Belgium, April 2006.