Performance Characterization of the FreeBSD Network Stack

Hyong-youb Kim and Scott Rixner
Department of Computer Science
Rice University
Houston, TX 77005
{hykim, rixner}@rice.edu
713-348-5060, 713-348-6353

Abstract

This paper analyzes the behavior of high-performance web servers along three axes: packet rate, number of connections, and communication latency. Modern, high-performance servers spend a significant fraction of time executing the network stack of the operating system—over 80% of the time for a web server. These servers must handle increasing packet rates, increasing numbers of connections, and the long round trip times of the Internet. Low overhead, non-statistical profiling shows that a large number of connections and long latencies degrade instruction throughput of the operating system network stack significantly. This degradation results from a dramatic increase in L2 cache capacity misses because the working set size of connection data structures grows in proportion to the number of connections and their reuse decreases as communication latency increases. For instance, L2 cache misses increase the number of cycles spent executing the TCP layer of the network stack by over 300% from 1312 cycles per packet to 5364. The obvious solutions of increasing the L2 cache size or using prefetching to reduce the number of misses are surprisingly ineffective.

Keywords: Web, TCP, Performance, Profile, Cache

1 Introduction

A modern, high-performance web server spends over 80% of its time within the network stack of the operating system. Therefore, the performance of the network stack influences the overall performance of modern web servers. However, the network stack's behavior is different than that of popular benchmarks that are typically used to measure processor performance. For example, on the same machine, the network stack achieves 0.2–0.3 instructions per cycle, whereas SPEC CPU2000 integer benchmarks achieve 0.7–1.0 instructions per cycle.

In order to understand this drop in efficiency, it is necessary to profile the network stack. Traditional profilers either do not include the operating system, since it can be invoked asynchronously by external events, or they interact with those asynchronous events and perturb the timing of the system. Such perturbations can result in significant changes in the overall behavior of the system. A low-overhead profiler that utilizes the processor's performance counters is needed to maintain the behavior of the web server while accurately monitoring its behavior.

This paper presents an analysis of the behavior of web servers along three axes: the packet rate, the number of connections, and the communication latency. A low overhead profiler is developed to examine the effects of each of these dimensions on web server performance. The experimental results show that overall system performance is largely unaffected by the packet rate of an individual connection. Rather, the number of connections and the communication latency of those connections have a significant impact on server performance. As the number of connections and the communication latency increase, the cycles consumed by the network stack per packet increases, while the instruction counts remain roughly same. This performance degradation occurs because of an increased number of L2 cache misses that occur while accessing connection data structures. Since modern DRAM latencies are several hundreds processor cycles, these L2 cache misses significantly reduce the processor instruction throughput. Detailed profiles indicate that the low achieved instruction throughput is almost entirely due to L2 cache misses.

The rest of this paper is organized as follows. Section 2 first describes the operation of modern web servers and the network stack. Section 3 describes the experimental testbed used throughout the paper. Sections 4 and 5 present an analysis of the performance of a web server and the network stack, respectively. Section 6 discusses the impact of L2 caches on performance and shows that increasing cache size and software prefetching are ineffective. Section 7 describes related work, and Section 8 concludes the paper.

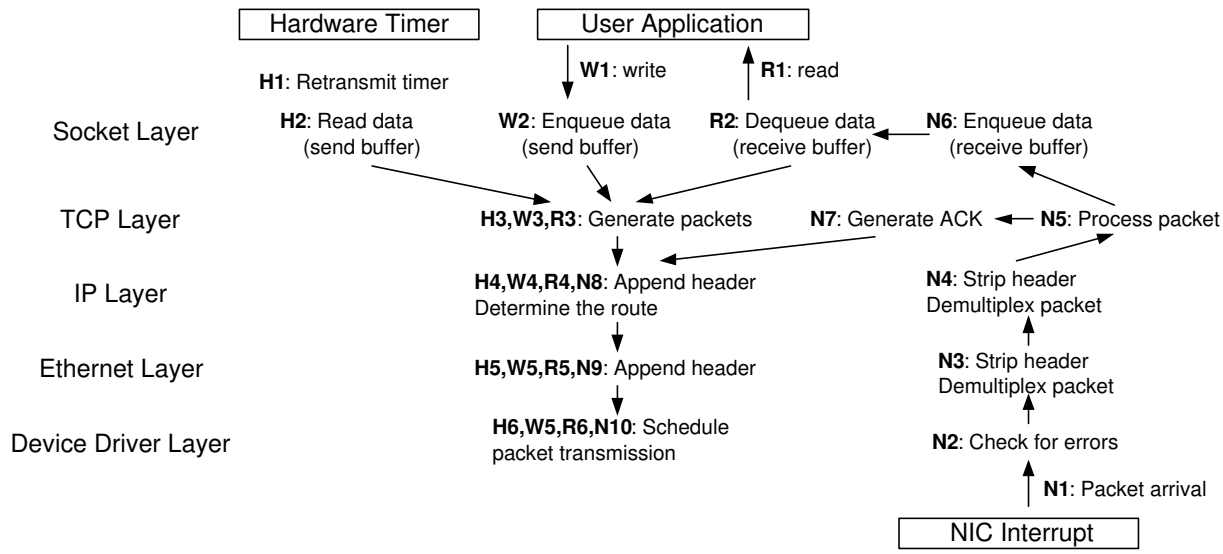

Figure 1: Tasks performed by the network subsystem in most operating systems assuming the layered implementation (network stack). The arrows show the flow of data within the operating system. The numbered steps of each flow have a distinct prefix according to its initiator. H, W, R, and N show the flows initiated by hardware timer interrupts, synchronous system calls (`write` and `read`), and hardware interrupts from the NIC, respectively. Only important tasks and data flow that occur most often are shown.

2 Modern Web Servers

A web server relies on the operating system to handle all messaging functions. Typically, a web server performs the following seven operations to satisfy a user request for static content (such as an image file):

1. Accept a new connection from a client.
2. Receive an HTTP request over the connection.
3. Parse the request.
4. Locate the requested file.
5. Generate the appropriate HTTP resonse header.
6. Send the header and file over the connection.
7. Close the connection.

Conceptually, these steps are quite simple, and a request for dynamic content would simply require the execution of a program or script to generate the response, rather than locating a file. However, a web server must be able to handle a large number of such incoming requests simultaneously. Furthermore, most of these operations require the use of operating system services, such as sending or receiving data, which belong to the network subsystem of the operating system.

Figure 1 shows how the layered implementation of the network subsystem (network stack) of the operating system can provide messaging services to an application, such as a web server, using the TCP/IP

protocol. The description is based on the FreeBSD 4.7 operating system, but should apply to other implementations based on the BSD network stack implementation. The figure shows five layers of the network stack. The socket layer is the interface to the user application which provides buffers to hold data to be sent out onto the network and data that has been received from the network. The TCP and IP layers implement the TCP/IP reliability, flow control, and routing protocols. The Ethernet layer implements the protocols of the Ethernet medium. Finally, the device driver communicates directly with the network interface.

Three types of events—system calls, NIC interrupts, and timer interrupts—can trigger actions in the network stack. Figure 1 shows how the network stack responds to these events. The middle of the figure shows how the operating system responds to the `write` and `read` system calls, assuming non-blocking sockets. The `write` system call can be used by the web server application to send response data over the network. In that case, the data will first be copied into a send buffer in the socket layer. The TCP layer will then generate TCP packets and will pass those packets to the IP layer. The IP layer will append an IP header to each packet and determine the route to send the packets. The Ethernet layer will then add an Ethernet header, including the MAC address of the next node along the route, to each packet and pass the packets to the device driver. Finally, the device driver will notify the NIC that there are new packets that need to be sent over the network. The `sendfile` system call can be used as an alternative to the `write` system call for zero-copy sends. `Sendfile` takes a file descriptor instead of the actual data, so data can be enqueued in the socket buffer by simply referring to it in the file cache, rather than copying it. All other steps remain the same as for `write`.

The `read` system call can similarly be used by the web server application to read request data from the network. The call first dequeues data from a receive buffer in the socket layer, and then the TCP layer may generate a packet to notify the client of changes in the receive buffer. If no data exists in the buffer, the `read` call will return without doing anything. The web server application accepts new connections by calling the `accept` system call. The TCP layer establishes a connection in response to a client's request up to some limit through the 3-way handshake sequence, independently of the application. So, the `accept` call itself usually involves modifying operating system data structures locally and does not generate packets.

When a packet arrives over the network, the NIC normally interrupts the CPU in order to notify the OS that a new packet is available. The right portion of Figure 1 shows how the operating system responds to these interrupts. First, the device driver checks for errors that may have occurred during the reception of the packet and translates the driver specific data structures that hold information about the packet, such

as the packet length, into the operating system specific data structures. The device driver then passes the packet to the Ethernet layer, which strips off the Ethernet header and determines which protocol was used to send the packet, such as IP or ARP, and passes it to the corresponding layer. For an IP packet, the IP layer similarly strips off the IP header and determines which protocol was used, such as TCP or UDP, and passes the packet to the corresponding layer. For a TCP packet that contains valid data, the TCP layer first processes the packet to determine its destination, then it enqueues the data into a receive buffer in the socket layer. In response to receiving the new data, the TCP layer generates an acknowledgment (ACK) packet. The acknowledgment packet is then sent to the IP layer, as if it were any other outgoing TCP packet. Finally, the application can access the data that was enqueued into the receive buffer in the socket layer using the `read` system call, as previously described. For a packet that contains an ACK, the TCP layer first removes the acknowledged data from the send buffer and may send one or more new packets, if the buffer still has pending data.

TCP is a reliable protocol, so if a packet is not acknowledged within a specified time period, the sender must retransmit the packet. The operating system uses hardware timer interrupts to determine if there are unacknowledged packets that need to be retransmitted, as shown on the left side of Figure 1. The TCP layer regenerates the original packets from the data in the send buffer. The new packets are then sent to the IP layer like all other outgoing TCP packets.

3 Experimental Testbed

Throughout this paper, measurements are taken on a testbed that consists of a PC-based server and two client machines. The server includes a single AMD Athlon XP 2600+ CPU, 4GB of DDR SDRAM, one 40GB IDE disk that stores programs, one 32GB SCSI disk that stores web files, and two Intel PRO/1000 MT Server Adapters (Gigabit Ethernet/PCI interface) on a 64-bit/66MHz PCI bus. The Intel NIC implements TCP/UDP checksum offload for both outgoing and incoming packets. Each client machine has a single AMD Athlon XP 2800+ CPU, 1GB RAM, a single 40GB IDE disk, and two Intel PRO/1000 MT Desktop Adapters. The machines are connected through two isolated Gigabit Ethernet switches such that each machine is on two subnets using two interfaces. The AMD Athlon XP 2600+ CPU used in the server has a unified L2 cache and separate L1 instruction and data caches. Each L1 cache is a two-way set associative 64 KB cache with 64 byte lines. The unified L2 cache is a 16-way set associative 256 KB cache with 64 byte lines.

All machines in the testbed run the FreeBSD 4.7 operating system. The server runs the Flash web server software. Flash is a multi-threaded, event-driven web server [14]. Flash also uses `sendfile` for zero-copy I/O, `kqueue` for scalable event notification, and helper threads for disk I/O. `Kqueue` is an efficient and scalable mechanism to notify an application when send socket buffers have space available for writing and when receive socket buffers have data available to be read [7]. The `kqueue` mechanism is more efficient than the traditional `select` system call, enabling flash to handle a larger number of simultaneous connections.

Each client machine runs two instances of a synthetic web client program which replays a given web access log. Each instance connects to the server through a different subnet. The client program opens multiple connections to the server to simulate multiple web clients and then generates web requests as fast as the server can handle them in order to determine the maximum sustainable server throughput. The traces are split equally among all four replayers. The web logs used for the experiments are from an IBM web site (IBM), a NASA web site (NASA), the Rice University Computer Science Department web site (CS), and the 1998 World Cup web site (WC). The NASA and WC traces are available from the Internet Traffic Archive. Requests from the same client (identified by IP address) that arrive within a fifteen-second period are issued using a single persistent connection. In addition to the four web traces, SPECweb99 (SPECWEB) is also used. Unlike the synthetic clients described above, SPECweb99 clients try to enforce a fixed bandwidth per connection (400Kb/s) by scheduling requests appropriately, and the working set size of files requested by the clients grows as the number of connections increases. SPECweb99 results are always based on runs that include the default mix of requests (70% static content and 30% dynamic content) and result in every connection achieving at least the default minimum required bandwidth (320Kb/s), unless otherwise noted.

Dummynet is used to increase the latency of communication between the clients and the server [15]. Since the Internet has orders of magnitude longer communication delays than systems sharing a single switch, it is important to simulate those delays using a mechanism like dummynet [13]. Finally, TCP's delayed ACK feature is disabled so that every sent segment is immediately acknowledged by the receiver; there are equal number of sent TCP segments and received ACK packets. With the delayed ACK enabled, the receiver may delay the generation of an ACK for a certain period in the hope that the ACK can be piggybacked onto a data segment. So, the ratio between sent segments and received ACK packets can vary depending on conditions. Disabling the feature allows for consistent performance analysis using a fixed ratio regardless of workloads.

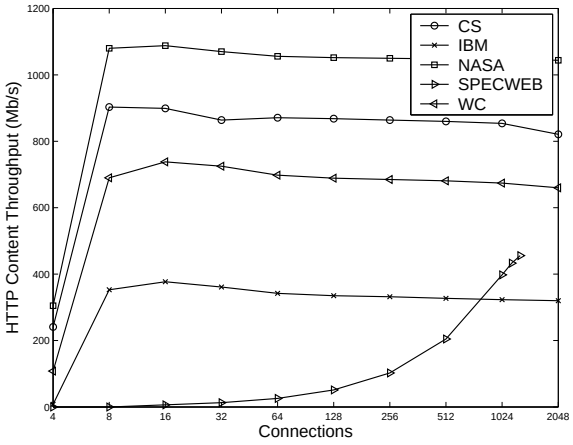

Figure 2: HTTP content throughput of web server achieved with varying number of connections for four web traces and SPECWEB.

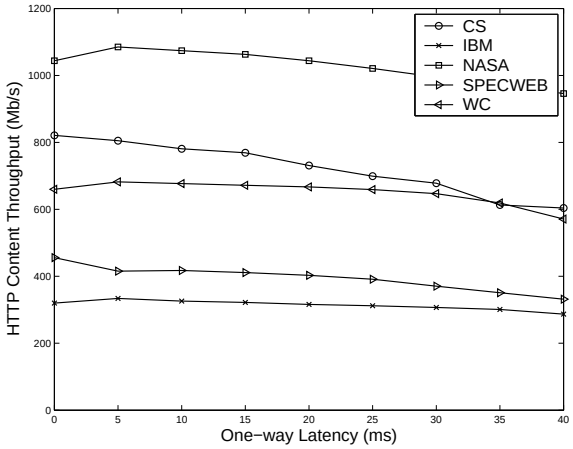

Figure 3: HTTP content throughput of web server achieved with varying network latencies for four web traces and SPECWEB.

The profiler developed for this study measures the execution time and processor performance statistics of individual kernel functions or groups of kernel functions. It provides a cycle-accurate measure of function execution time using the processor timestamp register that is incremented each clock cycle. In a manner similar to the Digital Continuous Profiling Infrastructure (DCPI), processor performance statistics are measured using hardware performance counters [1, 3, 6]. These performance counters count the occurrence of events such as retired instructions and data cache misses. The profiler aims to accurately measure these statistics while minimally perturbing the overall system behavior. The profiler achieves the former by computing its measures online, taking the dynamic control flow into account. It achieves the latter by profiling only those functions that are specified by the user. For the workloads used in this paper, the impact of profiling on system behavior is very small. The HTTP content throughputs achieved with profiling enabled are at most 5% lower than those achieved with profiling disabled. Profiling increases or decreases the number of various performance counter events such as L2 cache misses and TLB misses per packet by at most 6%, except the instruction counts. Profiling requires about 900 instructions per packet regardless of the workload. With profiling disabled, performance counter events are simply read once at the start and once at the end of an experiment. The difference between the two is the total events occurred during the experiment. With profiling enabled, events are accumulated for each profiled region, and the total events during an experiment equals the sum of events of all profiled regions.

Throughout this paper, the profiles are shown in the following code regions and categories. The code

regions are Driver, Ethernet, IP, TCP, and System Call. Driver is the device driver for the NIC. Ethernet consists of those functions that perform tasks related to the Ethernet layer. Likewise, IP and TCP consist of those functions that perform IP and TCP related tasks, respectively. System Call consists of all system call entries. Other, when used, consists of all other functions.

4 Web Server Performance

The performance of a modern web server is influenced by the number of simultaneous requests it must respond to and the round trip time of those connections. Figures 2 and 3 show the performance of the Flash web server as the number of simultaneous connections and network latency are varied. Figure 2 shows that as the number of connections increases, the server throughput first increases rapidly and then starts decreasing slowly. SPECWEB throughput increases at a much slower rate than the throughput of the four web traces because SPECWEB maintains a fixed bandwidth per connection. The maximum throughput of SPECWEB is reached using 1280 connections. Using more connections results in connections achieving less than the minimum required bandwidth. Figure 3 shows that increasing network latencies results in a similar effect. For the four web traces, 2048 connections are used. For SPECWEB, 1280 connections are used to see the impact of network latencies on the highest throughput achieved without artificial network delays. It was determined experimentally that realizable bandwidth of each connection drops below the minimum required bandwidth (320Kb/s) once network latencies increase beyond 25ms. Thus, SPECWEB runs result in connections achieving less than the minimum required bandwidth. A previous study on wide area network and web server performance shows similar trends [13].

As the server throughput degrades, the processor spends progressively more cycles per packet, while the instruction counts remain roughly constant as shown in Table 1. As a result, the instruction throughput deteriorates. The table only shows the profiles for the WC trace, but the other three traces all yield similar results.

As Table 1 shows, a modern web server can spend a significant fraction of its execution time in the operating system, over 80%. Moreover, processor efficiency, measured in instructions per cycle (IPC), is quite low. The average IPC of the network stack for WC using 2048 connections with 20ms latency is about 0.29 (TCP: 0.32, IP: 0.50, Ethernet: 0.39, Driver: 0.25). Similarly, the average IPC of the network stack for SPECWEB using 1280 connections with 20ms latency is about 0.24 (TCP: 0.29, IP: 0.54, Ethernet: 0.40,

		User	System Call	TCP	IP	Ethernet	Driver	Other	Overall Stack
WC									
(C:4)	Cycles (%)	2	2	3	1	1	3	5	10
(L:0)	Cycles	1582	1526	2509	571	1049	2064	3812	7720
	Instructions	574	499	1372	380	411	530	153	3292
(C:512)	Cycles (%)	15	15	29	5	10	21	6	80
(L:0)	Cycles	1861	1884	3663	676	1212	2689	746	10123
	Instructions	516	274	1349	377	412	612	49	3024
(C:2048)	Cycles (%)	15	15	28	5	9	21	6	79
(L:0)	Cycles	1993	2005	3743	690	1220	2741	787	10400
	Instructions	522	287	1376	377	412	616	57	3067
(C:2048)	Cycles (%)	14	15	33	6	8	19	3	81
(L:20)	Cycles	1817	1926	4277	740	1062	2397	408	10402
	Instructions	517	279	1377	373	410	587	42	3026
SPECWEB									
(C:16)	Cycles (%)	0	0	0	0	0	0	1	1
(L:0)	Cycles	3491	4107	2701	592	957	2136	9607	10493
	Instructions	1271	899	1414	393	396	780	376	3882
(C:512)	Cycles (%)	9	10	7	1	2	5	10	26
(L:0)	Cycles	3661	4141	3002	589	937	1987	4239	10657
	Instructions	1189	800	1407	385	396	702	192	3691
(C:1280)	Cycles (%)	24	28	21	3	5	12	7	69
(L:0)	Cycles	4191	4861	3639	613	947	2051	1300	12112
	Instructions	1124	603	1388	381	396	662	107	3430
(C:1280)	Cycles (%)	22	28	24	3	5	11	7	71
(L:20)	Cycles	4448	5567	4806	697	981	2247	1346	14298
	Instructions	1141	629	1397	377	396	660	103	3460

Table 1: Profiles of web server during the execution of the WC trace and SPECWEB. Cycles and Instructions show the counts per packet. C: and L: show the number of connections and latencies (milliseconds), respectively. L:0 means that the network latency is not altered artificially. Overall Stack accounts for all regions except User and Other.

Driver: 0.29). These compare very poorly against the IPCs achieved by some of the SPEC CPU2000 integer benchmarks measured on the same machine (vortex: 0.71, gzip: 0.81, eon: 0.89 crafty: 0.97). Furthermore, under better conditions, the network stack is able to achieve much higher IPCs (up to 0.92 for TCP), as will be shown later. Thus, the network stack inefficiently utilizes the processor as the number of connections and the latency of those connections increase.

5 Performance Analysis of the Network Stack

It is important to understand the cause of the inefficiency discovered in the previous section in order to improve overall system performance. This section analyzes the impacts of packet rates, connections, and network latencies on the network stack performance.

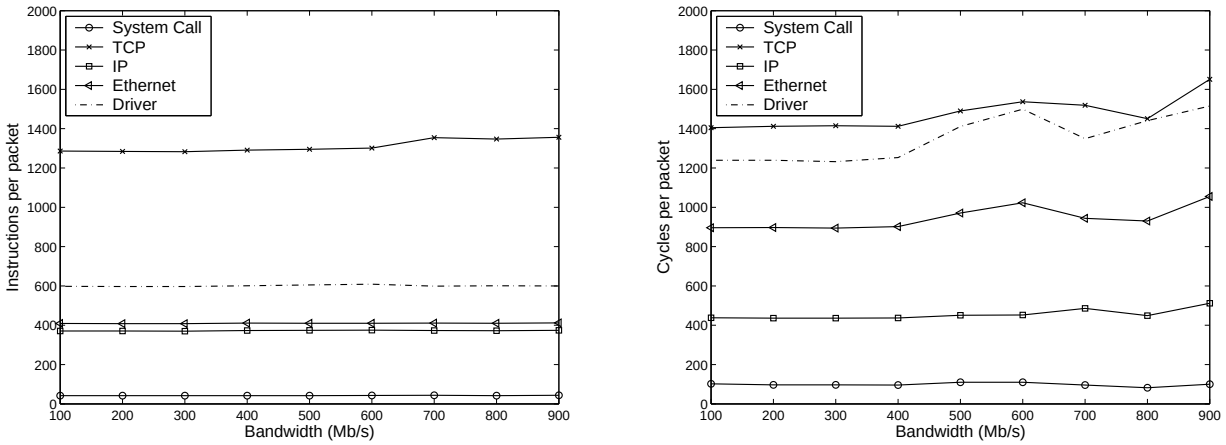

Figure 4: Instructions per packet (left) and cycles per packet (right) of the network stack for different network bandwidths over a single connection.

5.1 Packet Rates

Figure 4 shows how the achieved bandwidth of a single connection affects the number of instructions and cycles per packet of the network stack. This data was collected using a microbenchmark that opens a single TCP connection to another machine and sends maximum-sized (1460 byte) TCP segments at different rates. As the figure shows, there is very little change in the number of instructions executed per packet sent as the connection's bandwidth changes. The TCP layer's instruction count increases from 1286 to 1356 (by 5%) as the bandwidth increases from 100 Mb/s to 900 Mb/s. The instruction count per packet of the rest of the network stack remains very close to constant.

However, the cycle count per packet does increase slightly as the connection's bandwidth is increased. At 100 Mb/s, the IPCs of the System Call, TCP, and IP layers are 0.41, 0.92, and 0.85, respectively, and these layers incur no L2 cache misses. The IPCs of the Ethernet and Driver layers are only 0.46 and 0.48, respectively, as they incur L2 cache misses. As the connection's bandwidth is increased to 800 Mb/s, the IPC and the number of L2 cache misses remain largely unchanged for all of the layers. However, as the bandwidth is increased to 900 Mb/s, close to the line rate of the network, the cycle count per packet of the network stack increases from 4080 to 4833 (by 18%). The anomalous jump in cycle count around 600 Mb/s occurs largely because of the difficulty in precisely throttling the bandwidth of a single connection.

Note that at all bandwidth levels, the IPCs achieved by the network stack with this microbenchmark are much higher than the IPCs achieved by the network stack while running a web server, shown in Section 4.

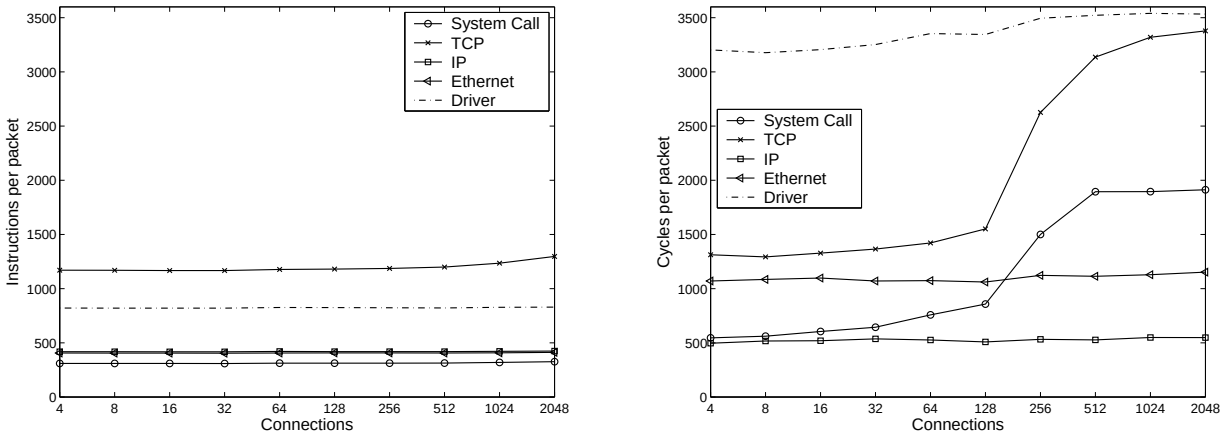

Figure 5: Instructions per packet (left) and cycles per packet (right) of the network stack for different numbers of connections with a constant total bandwidth of 100 Mb/s.

Overall, a connection's bandwidth does not appear to have a significant impact on the network stack's performance.

5.2 Connections

Figure 5 shows how the number of connections affects the number of instructions and cycles per packet of the network stack. This data was collected using a microbenchmark that opens the specified number of TCP connections to another machine and sends maximum-sized TCP segments at a constant rate of 100 Mb/s, divided equally across all of the connections. Therefore, the same amount of data is being sent, and the same number of acknowledgments are being received regardless of the number of connections. The low rate of 100 Mb/s is chosen to ensure that neither machine drops any packets.

Again, as the figure shows, there is very little change in the number of instructions executed per packet as the number of connections increases. The TCP layer's instruction count increases slightly from 1170 to 1297 (by 10%), as the number of connections increases from 4 to 2048. The instruction count per packet of the rest of the network stack remains very close to constant. However, as the number of connections increases, the cycles per packet of the System Call and TCP layers increase dramatically above about 128 connections. The cycles per packet also increase slowly for the Driver layer as the number of connections increases.

The increase in cycle count when the number of connections increases is caused by L2 cache misses. As a result of these misses, the IPCs of the System Call and TCP layers drop from 0.57 and 0.89 at 4 connections

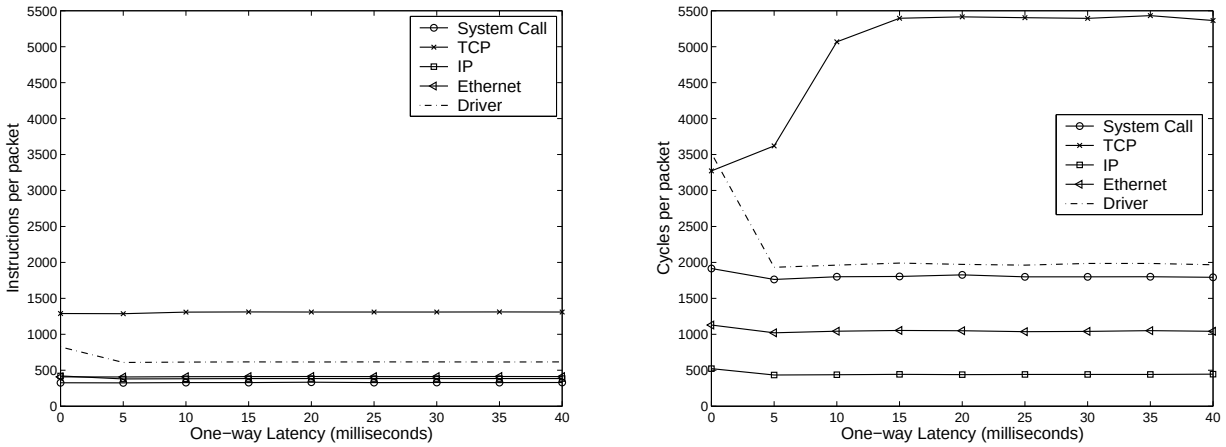

Figure 6: Instructions per packet (left) and cycles per packet (right) of the network stack as network latency increases for a fixed number of connections (2048) and a constant bandwidth (100Mb/s). 0ms latency means that the latency is not regulated artificially.

to 0.17 and 0.38 at 2048 connections. The average number of L2 cache misses per packet increase from none to 2.04 and 4.35 over the same interval. This results in an 157% increase in cycles per packet in the TCP layer and a 250% increase in cycles per packet in the System Call layer.

5.3 Latencies

Figure 6 shows how network latency affects the number of instructions and cycles per packet of the network stack. This data was collected using the same microbenchmark as in Section 5.2 using 2048 connections and a constant bandwidth of 100Mb/s spread evenly among those connections. However, the receiver machine used dummynet to add arbitrary latency to the communication between the two systems. The x-axes of the graphs in the figure show the one-way latency, which is identical in either direction.

For all cases in which dummynet is used to add additional communication latency, the number of instructions executed per packet across the network stack remains constant, except Driver. With increased latencies, the device driver sees lower interrupt rates, which changes the behavior of the driver. However, as network latency increases, the IPC of TCP drops from 0.39 (0ms; no dummynet) to 0.24 (40ms), while the L2 cache misses increase from 4.37 to 10.50. Over this interval, the cycle count of TCP almost doubles. The cycles per packet of the rest of the network stack remains fairly constant as network latency is increased.

6 Discussion

6.1 Impact of L2 Cache Misses

The operating system must keep track of each connection, which involves a file, a socket including socket buffer, a generic protocol control block, and a TCP control block. In order to send or receive a packet, as described in Figure 1, the network stack examines these data structures, takes the necessary actions, and updates the structures to reflect the latest state of the connection. Therefore, the working set of the network stack grows in proportion to the number of connections, and the most likely cause of the degradation of IPCs in response to increasing the number of connections is L2 cache capacity misses.

Similarly, the most likely cause of the degradation of IPCs in response to increasing latency is L2 cache misses due to the reduced locality of access to the connection data structures. It is perhaps non-intuitive that communication latencies should affect reuse of connection data structures. A client sends an ACK packet to the server upon receiving one or more TCP segments over a connection. During the time between the server sends a TCP segment and receives an ACK packet through a particular connection (round trip time), the server may send or receive packets through other connections. The communication latency then dictates the reuse distance of that connection measured as the number of connections that the server handles during the round trip time. Thus, given a large number of active connections, increasing communication latency also increases the reuse distance.

The main memory latency of the server system used to collect the data presented in this paper, consisting of a 2.1 GHz AMD Athlon processor with DDR266 SDRAM, is about 170 nanoseconds (measured using lmbench [11]). This translates to over 350 processor cycles, so even a small number of main memory accesses caused by cache misses could have a devastating impact on processor performance.

In order to confirm that L2 cache misses are the most significant problem within the network stack, Figure 7 shows the processor statistics that may account for the degrading IPC shown in Figure 5 when the number of connections increases. These are measured using the same machine and benchmark program used to generate Figure 5.

As shown in the figure, all statistics change very little as the number of connections increase, except L1 and L2 cache misses in the System Call and TCP layers. Remember that in Figure 5, the cycles per packet in these regions begin to increase at 32 connections and then sharply at 128 connections. Figure 7 shows that L1 and L2 cache misses per packet jump at 32 connections but L2 cache misses per packet still remain

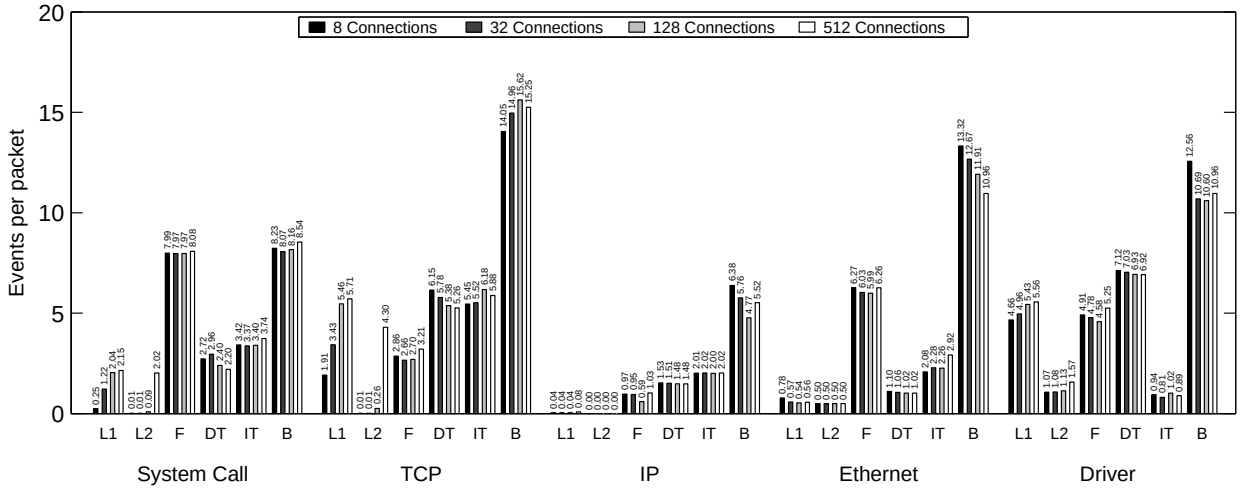

Figure 7: Statistics other than cycles and instructions per packet shown in Figure 5. The benchmark is the same as in that figure. L1, L2, F, DT, IT, and B show L1 cache misses, L2 cache misses, instruction fetch misses, data TLB misses, instruction TLB misses, and branch mispredictions per packet, respectively.

well below 1. In the System Call layer, L1 cache misses rise from 1.22 at 32 connections to 2.04 at 128 connections, while L2 cache misses increase from 0.01 to 0.09. The TCP layer shows similar results. L1 cache misses increase from 3.43 to 5.46, while L2 cache misses increase from 0.01 to 0.26. As the number of connection increases from 128 to 512, L1 cache misses increase only slightly but L2 cache misses increase dramatically. The L1 and L2 cache misses increase from 2.04, 0.09 in System Call and 5.46, 0.26 in TCP at 128 connections to 2.15, 2.02 in System Call and 5.71, 4.30 in TCP at 512 connections. These results indicate that (1) at 128 connections, the L1 cache can no longer store a significant fraction of connection data structures, (2) at 512 connections, the L2 cache's capacity also becomes too small to store connection data structures, and (3) the dramatic increase in cycles per packet seen at 512 connections is due to a surge in L2 cache misses. Thus, it can be inferred that main memory accesses due to L2 cache misses are indeed the biggest contributor to the degrading processor IPC.

6.2 L2 Cache Size

Increasing the cache size is a most common approach used by general-purpose processors in order to reduce cache misses. A larger cache would certainly be able to store more connection data structures. For instance, consider the following throughput and profile of the web server used so far in this paper with an AMD Athlon XP 2800+ CPU instead of an AMD Athlon XP 2600+ CPU. The former has a 512KB L2 cache, and the latter has a 256KB L2 cache. Both CPUs were run at the same frequency (2.1 GHz). Table 2 compares

		User	System Call	TCP	IP	Ethernet	Driver	Other	Overall Stack
2800	Cycles	1456	1464	2939	656	1190	2443	852	8693
(C:32)	Instructions	491	291	1359	379	404	613	54	3046
(L:0)	L2 cache misses	0.61	0.83	0.66	0.00	0.54	0.78	0.03	2.82
2600	Cycles	1627	1531	3063	621	1130	2458	798	8802
(C:32)	Instructions	490	290	1359	379	404	614	55	3047
(L:0)	L2 cache misses	0.99	1.20	1.23	0.01	0.54	1.30	0.03	4.28
2800	Cycles	1821	2107	4679	660	999	1980	397	10425
(C:2048)	Instructions	503	319	1373	375	403	590	46	3059
(L:30)	L2 cache misses	2.20	3.19	6.00	0.12	0.54	1.25	0.04	11.10
2600	Cycles	1956	2204	4742	735	1009	2129	387	10820
(C:2048)	Instructions	501	314	1373	375	403	590	46	3055
(L:30)	L2 cache misses	2.38	3.45	6.09	0.32	0.55	1.68	0.05	12.09

Table 2: Effect of doubling the L2 cache size (256KB vs. 512KB) while keeping the clock rate same. 2600 is the web server with an AMD Athlon XP 2600+ that has a 256KB L2 cache. 2800 is the web server with an AMD Athlon XP 2800+ that has a 512KB L2 cache but otherwise has the same configuration as AMD Athlon XP 2600+. The profiles are taken during the execution of the WC trace. Cycles, Instructions, and L2 cache misses show the respective counts per packet. C: and L: show the number of connections and latencies (milliseconds), respectively. L:0 means that the network latency is not altered artificially. Overall Stack accounts for all regions except User and Other.

the profiles of these two systems. With 32 connections, the 2800 incurs 2.82 L2 misses per packet in the network stack, as opposed to 4.28 L2 misses per packet on the 2600 (52% more). Accordingly, the 2800 has lower cycles per packet (8693 vs. 8802) and higher HTTP content throughput (719Mb/s vs. 704Mb/s). The improvement from a larger cache is only marginal. Increasing the number of connections and network latencies has similar effects on both systems as well. With 2048 connections and 30 millisecond one-way latency, cycles and L2 cache misses per packet increase to 11.10, 10425 on the 2800 and 12.09, 10820 on the 2600, and the 2600 now incurs only 9% more L2 cache misses. The HTTP content throughput decreases to 600Mb/s on the 2800 and 578Mb/s on the 2600. Again, the improvement from a larger cache is marginal. These results indicate that simply increasing the cache size would be an inefficient solution to improve the network stack performance given that the server needs to handle an increasing number of connections with long round trip times of the Internet.

6.3 Software Prefetching

When caching is ineffective, software prefetching is often used to reduce memory access latencies [5]. In order to determine whether prefetching can reduce the impact of L2 cache misses in the network stack, software prefetch instructions were manually inserted into the network stack code. In the device driver, a

		User	Sys. Call	TCP	IP	Ethernet	Driver	Other	Overall
No Prefetching	Cycles	1969	2095	4529	647	1048	2475	533	10794
564Mb/s	Instructions	502	29	1360	375	399	590	53	3024
	L2 cache misses	2.38	2.97	5.40	0.15	0.57	1.71	0.06	10.79
Prefetching	Cycles	1930	2307	4140	683	681	2546	537	10355
586Mb/s	Instructions	501	309	1376	375	399	588	52	3047
	L2 cache misses	2.37	3.78	4.88	0.21	0.07	2.28	0.06	11.21
				TCP		Ethernet			Fetch
Explicit Fetch	Cycles			3952		665			679
	Instructions			1405		399			109
	L2 cache misses			3.61		0.03			2.22

Table 3: Profile of web server during the execution of the WC trace with and without prefetching. Cycles, Instructions, and L2 cache misses show the counts per packet. Mb/s numbers show the HTTP content throughput achieved for a trace. Overall accounts for all regions except User and Other. In all cases, the server handles 5120 connections, and the client machines impose a 10 millisecond latency in each way. Explicit Fetch shows the profile of the code regions targeted by prefetching, when the prefetch targets are actually fetched using regular loads in a separate function Fetch.

single prefetch instruction is used to fetch the headers of a received packet when it processes the received packet. In the TCP layer, 6 prefetch instructions are used to fetch the protocol control block and the socket when it processes received packets. In the system call layer, 6 prefetch instructions are used to fetch the socket and the protocol control block when it sends data to the TCP layer. All of the above 13 instructions are `prefetcht0`, which is available on the AMD Athlon XP 2600+ CPU. These instructions are inserted as far in advance of the actual use of their target data as possible, without modifying the overall code structure. Several are inserted a function call ahead of the actual use of their target data.

Table 3 shows the effects of prefetching on web server performance. For brevity, the table shows the results using the WC trace alone, but the other traces result in similar behavior. As expected, the cycles per packet in the Ethernet and TCP layers decrease, by 367 and 389 cycles, respectively. The L2 cache misses in these regions show reductions as well. While L2 caches misses in Ethernet drop close to zero, the reduction in TCP is much smaller. This indicates that either the prefetches are useless or they are not early enough. Also, prefetching increases the cycles per packet in System Call and Driver, by 212 and 71 cycles, respectively. These increases are likely due to resource contention since the web server fully utilizes the CPU, and the network interfaces and the CPU compete for memory bandwidth. Overall, prefetching reduces cycles per packet in the network stack by 439 cycles and improves HTTP content throughput by 4% for the WC trace.

In order to find out whether the prefetches are early enough or are useless, the prefetch targets in TCP

and Ethernet are explicitly fetched (Explicit Fetch) in a separate function named Fetch. Since the data is explicitly fetched ahead of its use, the cycles and L2 cache misses in TCP and Ethernet represent the minimum achievable with prefetching. The cycles and L2 cache misses per packet in Ethernet with prefetching are already close to those with Explicit Fetch, showing that the prefetch in the device driver is early enough. However, Explicit Fetch yields much lower cycles and L2 cache misses per packet in TCP than prefetching. So, prefetching can potentially eliminate about 2 L2 cache misses if it executes earlier. Also, the remaining 3.61 misses mean that prefetching does not target all the misses that occur in TCP. Finally, Fetch incurs 2.22 L2 cache misses per packet, while TCP and Ethernet combined reduce L2 caches misses per packet by 2.33. Thus, useless prefetches are rare.

Software prefetching can potentially eliminate the remaining three L2 cache misses in TCP and further improve server throughput. However, it would be difficult to execute prefetch early enough to completely eliminate misses.

In addition to prefetching, one may attempt to increase spatial locality of data structures that span multiple cache lines by rearranging their fields. However, rearranging the fields of the connection control block structures (TCP control block and protocol independent control block) in most to least frequently accessed order results in no measurable impact on system performance. The access frequencies were gathered using a functional full system simulator [9], and the functional nature might have caused access patterns that are significantly different from real systems.

7 Related Work

Nahum et al. reported that caches have a big impact on the network protocol performance measured in latency [12]. Specifically, they show that larger, higher associative caches reduce latency, and that the protocol performance would scale with the processor performance as long as the caches provide the data necessary for the protocol processing. The findings of this study agree with the latter. However, larger caches would not necessarily improve the performance, when the number of connections is considered. Section 5 shows that the working set for the network stack grows proportionally to the number of connections. As the server handles an increasing number of clients (or connections), simply increasing the cache size would have limited benefits. A more recent study, performed by Makineni and Iyer, using a real machine also reports that cache misses can significantly affect the network stack performance [10]. However, they still ignore the

impact of a large number of connections and long latencies that servers experience and consequently do not investigate their negative impact on performance.

Luo et al. evaluated a number of server workloads, including SPECweb99, using three different real systems [8]. They show that web servers spend a large fraction of time executing the kernel and in general achieve lower IPCs than the SPEC CPU2000 integer benchmarks. They also report frequent L2 cache misses on web servers and their noticeable impact on IPC. Finally, they compare different L2 cache sizes (512KB, 4MB, and 8MB) and show that even large L2 caches are not able to capture working sets. These findings generally agree with the results presented in this paper. However, this paper relates the number of connections and communication latencies to server performance and identifies connection data structures as a major cause for L2 cache misses. Some researchers used a full system simulator and also found that commercial workloads like web servers cause frequent L2 cache misses [2].

Digital Continuous Profiling Infrastructure (DCPI) first used performance counters for profiling [3]. DCPI is a statistical system-wide profiler. It can profile both user and operating system code and accurately attribute counter events such as cache misses to the instructions that cause them. It reportedly incurs very low overhead and is thus suitable for profiling the operating system. However, it only supports the Tru64 operating system running on Alpha processors, which led the authors to develop the profiler for the purpose of this study. Following DCPI, a number of profilers have recently appeared that make use of performance counters. These include Intel's VTune and various profilers based on Performance Application Programming Interface (PAPI) [4]. They primarily target application profiling, and it is unclear whether they are suitable for profiling the operating system at the level of detail presented in this paper, without significantly disrupting the system behavior.

8 Conclusions

This paper analyzes web server performance along three axes: the packet rate, the number of connections, and the communication latency. Experimental results based on both microbenchmarks and web server workloads reveal several key findings. First, the packet rate of a connection has a negligible effect on the performance of the network stack, as the average number of cycles per packet within the network stack remain somewhat constant. Second, increasing the number of connections increases the number of L2 cache misses within the network stack. This increases the number of cycles per packet and decreases the achieved IPC.

The highest IPC is achieved using a single connection. Third, increasing the network latency also increases the number of L2 cache misses. Again, this increases the number of cycles per packet and decreases the achieved IPC.

The increase in L2 cache misses are caused by the increased working set size of connection data structures, including TCP control blocks and socket structures. High network latencies further decreases reuse of these structures. These cache misses increase the time spent within the TCP layer of the network stack by over 300%, resulting in a 73% decrease in achieved IPC. Neither increasing the size of the L2 cache nor prefetching the connection data structures within the network stack result in significant performance improvements under these conditions. The former is inefficient because the working set size of a large number of connections can easily exceed even largest L2 caches available today (a few megabytes). The latter is ineffective because prefetching data hundreds of processor cycles ahead of its use in order to overlap computation and memory access latency is very difficult. Therefore, new solutions are needed in order to improve the performance of modern web servers.

References

[1] Advanced Micro Devices. *AMD Athlon Processor x86 Code Optimization Guide*, Feb. 2002. Revision K.

[2] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proceedings of the 5th Workshop On Computer Architecture Evaluation using Commercial Workloads*, Feb. 2002.

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 1–14. ACM Press, 1997.

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52. ACM Press, 1991.

[6] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2002.

[7] J. Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[8] Y. Luo, J. Rubio, L. K. John, P. Seshadri, and A. Mericas. Benchmarking Internet Servers on Superscalar Machines. *Computer*, 36(2):34–40, Feb. 2003.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hállberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[10] S. Makineni and R. Iyer. Architectural Characterization of TCP/IP Packet Processing on the Pentium M Microprocessor. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 152–162, Feb. 2004.

[11] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Technical Conference*, pages 279–295, January 1996.

[12] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Cache Behavior of Network Protocols. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 169–180. ACM Press, 1997.

[13] E. M. Nahum, M.-C. Rosu, S. Seshan, and J. Almeida. The Effects of Wide-Area Conditions on WWW Server Performance. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 257–267. ACM Press, 2001.

[14] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 199–212, June 1999.

[15] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.