Optimizing TCP Forwarding

Vsevolod V. Panteleenko and Vincent W. Freeh

TR-02-03

Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
{vvp, vin}@cse.nd.edu

Abstract—*The continued growth of the web places ever increasing performance demands on web site front-end appliances. In many cases, these appliances have to forward network traffic to and from web servers at transport and application levels utilizing complete TCP/IP stack processing, which could easily make the front-end appliance a bottleneck for a web site.*

This paper describes four novel optimizations of the TCP/IP stack processing for a TCP forwarding appliance: acknowledgement aggregation, fast path for incoming packets, double allocation avoidance in TCP module, and packet reuse. These optimizations are applicable for different throughputs and MTU sizes on the forwarding path when traditional approaches for performance improvements, such as TCP splicing, could not be applied. These optimizations are implemented in context of a web booster appliance that modifies web site traffic to decrease the cost of the network processing on a web server. These four optimizations, applied together with device driver polling, result in four-fold improvement in appliance throughput compared to a base case.

I. Introduction

High-volume public web sites use a complex infrastructure with functionality partitioned among multiple tiers. Most such sites deploy front-end appliances that handle client requests in some way before delivering them to the web servers. Functionality of these appliances ranges from load balancing of the web servers to encryption/decryption offloading. In many cases the appliances have to accomplish processing at the transport and application levels by terminating client TCP connections and forwarding data to the new connections opened to the web servers, potentially performing some processing of the data. Full TCP/IP processing of the client connections is an expensive operation [29] and the TCP forwarding appliances can easily become a bottleneck for a web site.

This work was sponsored by IBM University Partnership Program Award.

This paper describes optimizations of TCP/IP stack processing for TCP forwarding appliances. They are developed for the common case of HTTP server network traffic. By optimizing packet send and receive processing, the appliance can achieve a four-fold increase in throughput over the base case. The optimizations are implemented and evaluated in context of a novel appliance, the web booster, which decreases the overhead of the network processing on the overloaded web server. This is accomplished by changing network characteristics of the client requests before forwarding them to a web server. While these optimizations are developed and implemented in the web booster, they may be applied to other types of TCP forwarding proxies. Some are directly applicable to a web server, if it is permissible to modify the server's network protocol stack and operating system.

The web booster uses five optimizations that improve the performance of TCP/IP processing at the web booster. The optimizations are: (i) incoming acknowledgement aggregation, (ii) fast path for incoming packets, (iii) double allocation avoidance in TCP module, (iv) packet reuse, and (v) device driver polling. The first four are developed for the booster; the last optimization is previously known [11][23]. The optimizations require modifications to the network protocol stack, device driver, and operating system of the web booster to streamline interaction among these components.

Unlike most TCP forwarding appliances, the web booster terminates the incoming TCP connections and buffers data in order to support different data rates and connection MTU sizes on the forwarding path. Different data rates allow short connection duration on one side of the forward path even if the other side has limited bandwidth. The web booster also supports connection multiplexing, when several client connections are multiplexed into one server connection using HTTP level information. This design makes it impossible to utilize well-known optimizations of the network protocol processing of TCP proxies, such as TCP splicing [20].

This paper is organized as follows. Next section describes related work. Section III outlines the web booster architecture. Section IV describes design and implementation of the web booster. Section V introduces four novel processing optimizations. The description of the test environment used for evaluating the optimizations is given in Section VI. Section VII measures the effect of the optimizations on performance of the web booster. Finally, Section VIII concludes.

II. Related Work

There are many commercial and research appliances that perform packet forwarding using TCP and HTTP level information, such as content-based routers [1][3]. The layer at which packets are forwarded depends on the HTTP request distribution policy. Weighted round-robin distribution policy requires only TCP (layer-4) forwarding [12][15]. Because server performance is highly dependent on the memory cache hit rate, forwarding requests to a server that has already cached the resource in memory provides significant performance savings. Such content-based distribution [3][27] requires HTTP (layer-7) forwarding support. Because the web booster performs request multiplexing, it also requires layer-7 forwarding support.

TCP splicing is a widely used method for improving the performance of TCP forwarding [13][20][35][38]. It virtually terminates the connections at the forwarding host, but physically it only modifies the forwarded packet IP address, TCP port, and some other information. TCP splicing is not applicable for the web booster because the booster has to build new packets before forwarding data to support different data rates and MTU sizes on forwarding path.

TCP hand-off enables forwarding of the server responses directly to the clients without passing them through TCP forwarding appliance [4][27]. Aron et al. [5] suggested distributing content-based forwarding by offloading the initial TCP connection establishment to the web server node in a cluster. This node further redirects the connection based on the content. In this case, the front-end TCP router does not have to be a content-based (layer-7) router.

There are several commercial appliances which modify the network properties of the client request stream to offload the web servers [8][26][32]. They are similar in purpose to the web booster. Unfortunately, there are no details of the design and implementation for these appliances publicly available.

There is a body of research related to network protocol processing optimizations for general-purpose systems. Header prediction algorithm [16] provides a fast path for mostly unidirectional network traffic and has been incorporated in majority of TCP/IP implementations. Several studies reduce data copying throughout the protocol stack and the number of interrupts that are required to process packets [9][17][31]. Other optimizations include efficient incoming packet de-multiplexing [18], implementing one-behind cache for incoming packets [31], and improving hardware cache behavior by rearranging data and code [24].

There are multiple studies that analyze the performance of web servers and the effects of different design choices. Some of the performance studies use a simulated workload [6][14], while others measure server behavior under real-life workload [21][22][34]. Pai et al. looked into performance of different server architectures in context of one implementation: the Flash web server [28]. Lava caching server shows the effect of using a specialized operating system and building the server as integrated software [19]. Nahum et al. [25] studied the effect of a WAN environment on the performance of a web server. Panteleenko and Freeh [30] analyzed internal operation of a web server under simulated WAN workload. These studies show the importance of reproducing such workload characteristics as request inter-arrival time, large number of simultaneously open connections and WAN network conditions.

III. WEB BOOSTER ARCHITECTURE

This section overviews the web booster architecture, consisting of the web booster appliance and an accelerator software module located on the web server. The section gives only the necessary background for understanding the context in which network protocol processing of the booster appliance occurs. A detailed description and performance evaluation of the booster architecture can be found elsewhere [30].

The web booster architecture is used for reducing the load of a web server during peak traffic. The architecture instantaneously decreases the processing cost on a web server for requests to both static and dynamic web resources. During a period of peak traffic, client requests are redirected to a web booster, which is located in front of the web server, that changes that network characteristics of the requests and forwards them to the web server in an optimized request stream. The web server processes this modified request stream at a significantly reduced cost compared to processing of an unmodified stream. The offloading is achieved without modifying the site infrastructure or changing the server software other than adding the accelerator module. The fact that the load may be decreased instantaneously enables effective sharing of the web booster among multiple web servers that have a low correlation of load peaks.

The web booster reduces the network processing cost on the web server. This cost is a majority of the overall overhead for static and many types of dynamic resource processing [29]. The overhead is reduced by decreasing the number of packets processed by the server and by reducing the cost of packet processing. Decreasing the number of packets processed is accomplished by (i) increasing the MTU of the incoming packets, (ii) avoiding TCP connection open and close, and (iii) delaying TCP acknowledgements to the web server. The web accelerator software module running on the web server further decreases the cost of the packet processing by (i) performing HTTP request processing for static documents completely in the kernel mode, (ii) avoiding copying and checksumming for sent data, and (iii) avoiding packet allocation cost. Implementing the web accelerator as a kernel module enables the first optimization. A packet cache, which stores the responses in the form of TCP packets, enables the other two optimizations. These techniques can decrease the web server load by nearly an order of magnitude [30].

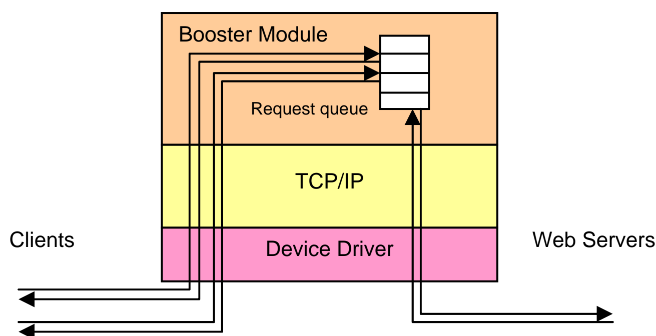

Figure 1. Web booster design

IV. WEB BOOSTER DESIGN

The web booster sits between clients and servers, terminating client TCP connections. It forwards client data on different connections open to the servers and sends data received on the server connections back to the clients using the original client connections. The web booster supports connection multiplexing, in which several client connections are multiplexed into one server connection using application (HTTP) level information. It also supports different data rates and MTU sizes on the client and the server connections in order to decrease the network protocol processing overhead on the web servers. This is accomplished using buffering at the booster. Different data rates allow for sending the data on the server connections with maximum connection throughput, even if the client connections have limited bandwidth and require a long time to transfer the received data.

At a high level, the web booster consists of several components (Figure 1). The device driver and the TCP/IP stack provide incoming and outgoing data streams to a booster module. The booster module is responsible for multiplexing client connections and handling the HTTP-level semantics of request forwarding. It maintains a pool of permanently opened connections to the server, through which client connections are multiplexed. To enable multiplexing, the booster module changes the HTTP headers of the incoming request to be HTTP/1.1 headers with the persistent connection option. Response headers from the server to clients are modified by the booster to correspond to the original client requests.

The booster module uses a request handle to track each incoming client request. A handle references the client socket and aids in de-multiplexing server responses. When the booster forwards the client request to the server, a request handle is queued at the server socket (using a special handle queue implemented in the booster module). When a response arrives from the server, the handle at the head of the queue identifies the client and its socket. Thus the booster keeps an association between multiple pipelined requests and responses sent on the same connection, information not available in the HTTP responses.

The web booster has to perform the following tasks to process client requests. It listens on a TCP port, accepts a connection request from a client and allocates a request handle. After the HTTP request is read from the accepted connection, the HTTP request header is modified. The request is forwarded to the server using one of the permanently opened server connections, and the request handle is queued at the tail of this connection's handle queue. The response data are read from the server connection in the order the requests were sent (which is specified by HTTP/1.1), and a request handle located at the head of the queue is used to identify the request. The HTTP header of the response is modified to correspond to the original HTTP request header, and the data are forwarded to the client on the original client connection. This design allows pipelining the requests sent to the server with the responses from the server.

Client and server connections usually have different throughput. The low throughput of a client connection might stall other responses that arrive on the same server connection. The web booster buffers responses so that the server connections can transfer data at a full speed. The send queues of the client

sockets are used as buffers. The data packets are queued at the client sockets as soon as they arrive from the server.

Very large documents might overflow the booster memory if the throughput to the client is low. This would effectively disable the transfer of all other documents which are potentially destined for other clients. Therefore, the booster uses a cut-off size for buffering documents. If the size of the document is above some threshold, the TCP connection used for receiving the response is removed from the pool of permanently open server connections. It is used exclusively to transfer this file at the client speed, and it is closed after the transfer. At the same time a new connection to the server is opened and added to the connection pool to replace the old one. All requests that were issued at the old connection after the current request and are pending for processing by the server are reissued at the new connection. Thus large documents do not block permanently opened server connections or consume large amounts of memory on the booster.

The web booster is implemented using a general-purpose operating system (Linux 2.4). The operating system and the network device driver are changed to use polling of the device driver instead of using hardware and software interrupts [11][23]. The device layer is also modified to streamline passing of the received packets to the protocol layer. Received packets are directly passed to the protocol level without queuing them at the general backlog queue. In the unmodified device layer, the device driver accomplishes such queuing in a hardware interrupt context, and the queued packets were further processed in software interrupts.

The booster code responsible for HTTP processing and connection handling is implemented as a kernel module. This module also provides a scheduler that is responsible for fair processing of the events on different TCP connections generated by received packets and timers. The booster avoids data copying and checksumming by relying on the hardware capabilities of the network adapters, particularly on scatter/gather buffers and hardware checksumming. The packets received from the server as the HTTP response are used in-place to build new outgoing packets by allocating a packet header and pointing to multiple discontiguous data segments in the received packets. These new packets are queued at the client socket send queue.

V. Network Protocol Optimizations

This section describes four novel techniques that significantly improve the performance of the web booster network processing. These techniques modify the TCP/IP stack to improve the common case for network processing in the booster environment. Unlike many TCP/IP optimizations that speed-up the common case for the general network processing, such as the header prediction algorithm [16], the web booster techniques address specifics of the network processing for HTTP traffic. If incorporated in a general-purpose operating system, these optimizations could increase the overhead of the processing by performing unnecessary additional computations when network traffic would not correspond to the expected common case, or they could introduce high network latency. However, in the context of specialized booster processing only, these optimizations pose no problems. Some of these optimizations can also be applied to the web server if modifications of the server's operating system are allowed.

The first two techniques optimize the receive path, and the last two deal with packet sending. They are as follows:

- Acknowledgement aggregation,
- Fast path for packet receive,
- Duplicate allocation avoidance on the send path, and
- Sent packet reuse.

The rest of the section describes these techniques in detail.

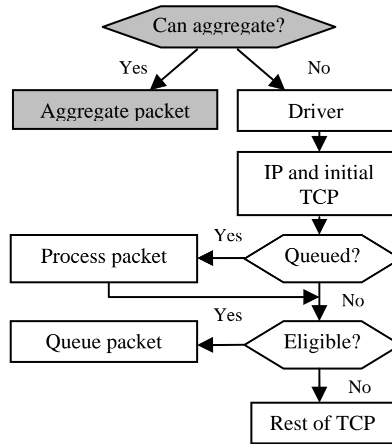

Figure 2. Booster acknowledgement aggregation

A. Acknowledgement Aggregation

The first optimization uses acknowledgment aggregation. This optimization decreases the number of received packets that are fully processed by eliminating some acknowledgment packets. Aggregation is based on the observation that after connection establishment and receiving a client request, most of the packets from a client to booster are acknowledgement packets with no data, and the acknowledged sequence number is strictly monotonically increasing. For 10KB file size, which is close to the average size of HTTP objects [7], number of such packets is significantly higher than the number of other types of packets. Often it is possible to combine several consecutive acknowledgments into one at an early stage in processing, and later process them as one packet. This design not only reduces acknowledgement processing, it also saves the packet allocation and freeing cost for aggregated packets.

Figure 2 shows the aggregation algorithm. Processing of a received packet can be delayed by queuing it at a TCP socket before performing full state-specific processing. This packet becomes the base for aggregation and is used to hold the data from the packets aggregated with it. The check performed for delaying the packet is depicted as "Eligible?" box on Figure 2. During the device driver receive operation, every packet is checked to see if it can be aggregated with the packet queued at the socket, if any. If the received packet passes the check, the two are aggregated: the queued packet is updated with the new sequence number and potentially with the new advertised window. The device driver can immediately reuse the received packet. This path is shaded gray on Figure 2. If, on the other hand, the received packet fails the check, it is delivered to the socket in a regular way, using the path depicted as two boxes "Driver" and "IP and initial TCP" on Figure 2. Before this packet is processed, the base packet previously queued at the socket is processed first.

Aggregation is used only when the receiving socket is in FIN_WAIT_1 state, because the majority of the packets received by the booster are processed in this state. The response from the server arrives over a high-speed LAN connection with large MTU, so it gets queued on the client socket rather that sent to the client due to the TCP slow start. After the response is received completely, the booster closes the client connection, which changes its state to FIN_WAIT_1. The rest of client packets, which are client acknowledgements, are therefore processed in the FIN_WAIT_1 state. They open the congestion window and send queued data. The number of packets received form the server, which are processed in the ESTABLISHED state, is significantly smaller than the number of client packets due to the small MTU size of the client connections compared to those of server.

In order to be aggregated, packets have to satisfy the following requirements so that aggregation would not violate TCP.

- The packet has to be a pure acknowledgment: no data and only ACK flag set.

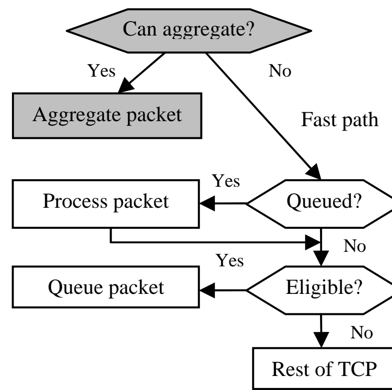

Figure 3. Fast path for packet receiving in booster

- The only TCP option that is allowed in the packet is timestamp. This guarantees that there are no selective acknowledgements in the packet that have to be processed in a general way.
- The packet has to increase the acknowledged sequence number for the socket. This avoids delaying duplicate acknowledgments which are critical for the fast retransmission algorithm.
- The socket sequence number for the next data to be sent has to be greater than the acknowledgment number. This guarantees that there are sent data that are not acknowledged yet and the booster expects these acknowledgments to arrive later. Otherwise, the delayed acknowledgement can be the only one that has to open the congestion window and the socket can be stalled because no data can be sent before the window opens.
- The socket has to be in FIN_WAIT_1 state and it cannot be retransmitting (for selective acknowledgment state machine, it has to be in OPEN state). This is necessary because in other states the received acknowledgment requires additional processing.
- The number of successively aggregated acknowledgments should be less than some maximum number. This limits the delay possible.

Arriving acknowledgements control the congestion window for a socket. Without this optimization, every received acknowledgment increases the window by one segment. This optimization makes the increase more coarse-grained: the congestion window is increased by several segments at once when the aggregated acknowledgment arrives. Nevertheless, it does not violate the protocol: conditions for the aggregation guarantee that the acknowledgment is not delayed when it is critical for opening the window. It is delayed only in the case when the following acknowledgments that open the window are expected to arrive.

The fast retransmission algorithm is driven by duplicate acknowledgments. TCP specifies that when an out-of-order packet is received, the TCP module has to immediately send the acknowledgment for the last correctly received packet, which in this case, is a duplicate one. Such duplicate acknowledgments are not delayed with this optimization, enabling the fast retransmission algorithm to work correctly.

Acknowledgement aggregation is performed in the driver receive routine. Before a packet can be aggregated, it has to be checked to be a valid TCP packet and the socket for the TCP port specified by the packet has to be found. These operations have to be performed for both acknowledgment aggregation and the fast-path optimization, and are described next.

B. Fast Path for Packet Receiving

The second optimization streamlines the initial TCP/IP packet processing in the common case for web booster network traffic. Such traffic has the following two properties: most of the arrived packets are TCP

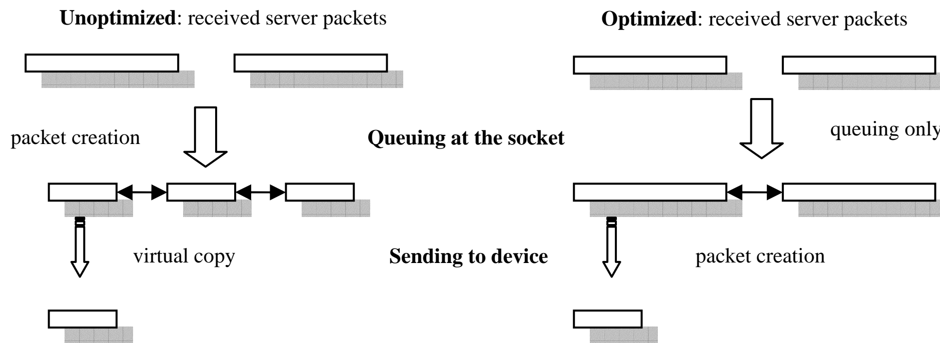

Figure 4. Duplicate allocation avoidance in booster

packets for already open connections and the IP routing for most of incoming packets is simply delivering them to the upper TCP protocol. In this case, it is possible to perform several simple checks on an incoming packet early in the processing and post the packet that passed the check directly for the state specific TCP processing. This avoids processing paths in the device driver, IP module, and initial TCP processing. If a packet fails the check, it is processed in the regular way. The checks that are performed on the packet include TCP/IP header size verification, checking for absence of IP options (common case), and lookup for the TCP socket. The latter avoids a routing table lookup because the socket entry in the hash table contains the necessary IP information to verify that the packet has to be delivered to the upper protocol. The network adapter performs the IP and TCP checksum verification.

The fast-path checks are performed as a direct call from the driver receive routine. These checks mostly overlap with those of the previously described acknowledgment aggregation optimization and are performed together. In this case, a packet is either aggregated, or use fast path before state specific processing (as shown on Figure 3), or take the regular path for small fraction of received packets (not shown on the figure). The fast path optimization may be considered an extension of acknowledgement aggregation.

C. Duplicate Allocation Avoidance on the Send Path

The third optimization applies to the data send operation. It avoids double allocation and freeing of the packets to be sent. When the data are written to a socket in a standard TCP implementation, the packets are built and queued at the socket send queue, even if the actual data send operation takes place later (Figure 4). When the data are ready to be sent, for example when the congestion window opens, these packets stay at the write queue for further possible retransmission and are freed only when the data are acknowledged. Instead of sending a queued packet, a new virtual copy of the packet is built by allocating a new packet header, pointing to a packet data and incrementing the reference count of the referenced packet data (in Linux it is called “cloning”). This new packet is passed to a device driver and, when the send operation finished, is freed by the driver.

Instead of building sent packets twice, the optimized TCP module builds the packets only once during the actual send operation. The socket write queue is used to hold directly the packets received from a socket opened to the server: there is no packet creation at the time of queuing. Although these packets cannot be used directly for virtual copying during send operation (because they usually have a larger MSS), they are used as data buffers. The packets to be sent are built by allocating the header and pointing to potentially multiple discontiguous data regions in those packets queued at the socket write queue. The latter are freed when the data get acknowledged. In this way, one allocation and one freeing operation are saved for every sent packet. In addition to that, the process of freeing the packets on the socket write queue is less expensive because the size of the packets is usually larger than in the un-optimized case and there are fewer of them to search and de-allocate.

D. Sent Packet Reuse

The fourth optimization reuses sent packets for data transmission. Instead of de-allocating packets after the driver completes the send operation, freed packets are queued at the special queue for a socket. During the packet send operation, these queued packets are reused instead of allocating and building new packets for the buffered data. In this case, the headers already contain most of the valid TCP/IP information. Necessary updates include pointing to the new data buffers and updating TCP sequence numbers. After that, the packets are queued at the device queue using the fast IP path to avoid IP header building. The number of packets that have to be queued for reuse is small because in most cases the time the packet is used for the send operation by the device driver is significantly shorter than the time between successive send operations triggered by arrived acknowledgments.

VI. Experimental Methodology

The experiments described in this paper use several tools that emulate client workload. The first one was written for micro-benchmarking. It requests a single object from the server with the constant request rate.

The second tool is used for generating realistic client workload. It was built from the SURGE workload generation tool [7]. SURGE emulates the statistical distribution of the following real-life workload parameters: (i) *server object size* distribution using lognormal model for the body of the distribution and Pareto model for the tail, (ii) *request size* distribution using Pareto model, (iii) *relative object popularity* using Zipf model, (iv) *embedded object references* using Pareto model, (v) *temporal locality of references* using lognormal model, and (vi) *idle periods* of individual users using Pareto model. It has been shown that SURGE emulates real-life characteristics of client requests better than the standard tools, such as SPECWeb [36], particularly self-similarity of the server network traffic produced by the requests [7].

SURGE was modified for these experiments in two ways. First, the process-based design was changed to an event-based design in order to increase the maximum number of simulated clients. Second, the server throttling problem [6] present in the original SURGE design was eliminated. Server throttling occurs in an overloaded server when the higher processing latency of the requests delays the next request generated by the emulation tool, which waits for the completion of the previous request, and thus effectively decreases the request rate of the simulation. The modified SURGE does not wait for the current request completion to calculate the time for the issuing of the next request.

The third tool that was written for these experiments emulates network delays and bandwidth limits of the client connections. It is based on the Linux kernel and its TCP/IP stack. The receive path of the network protocol stack was modified to delay packets in order to emulate the network parameters. Such packets are put on the queues that are distinct for each TCP socket. Standard Linux timers schedule the delivery or the transmission of the delayed packets with 10 ms granularity. Similar experiments [25] that used the Dummynet tool [33] indicate that 10ms granularity is precise enough to emulate network effects for the web server performance studies. Unlike these studies, which can only emulate network characteristics per network interface, our tool allows emulating parameters for each TCP connection. This is necessary in order to emulate bandwidth limit to clients. Tools like Dummynet and NISTnet [10] cannot do this because the bandwidth bottleneck is usually near the client and not at the network link to the server. The current limitation of our tool is that it does not emulate of packet losses and reordering.

The fourth tool performs web booster profiling. It measures time spent in different parts of the Linux kernel using CPU hardware counters. This tool uses breakpoints in the code to measure the time spent in a particular code path including all functions invoked in this path. Profiling is suspended during context switches to a different process or during hardware or software interrupts if the profiling was started in a process context. This method allows measuring the time spent in all call sub-graphs of interest, not just time spent in a particular function regardless of the source of its invocation, as statistical sampling tools do [2]. The overhead of such profiling was measured by setting a dummy pair of breakpoints to start and stop the timer. These breakpoints were set in the actual code to measure such effects as processor cache

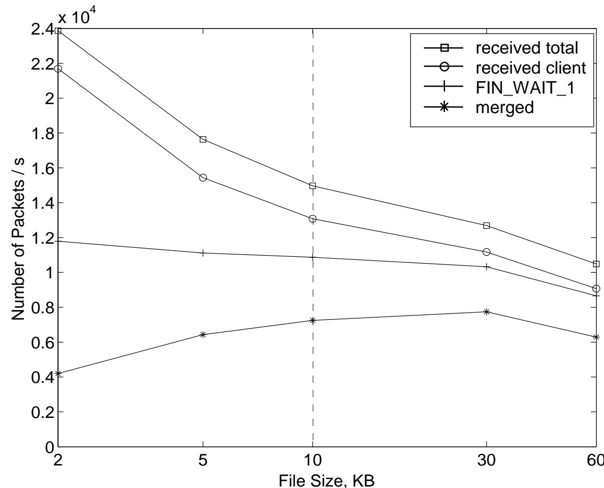

Figure 5. Number of packets vs. file size

misses between the successive breakpoint invocations. At a rate of about 20,000 invocations of this breakpoint pair per second, the total overhead of profiling was about 0.08% of the total time spend in processing.

Experiments use one web server, one web booster and two client machines of the same configuration. The machines are Intel PIII 650MHz with 512MB of main memory running Linux 2.4 kernel. Each client is connected by one 3COM 3C590 100 Mbps network adapter to the web booster with the same type of the network adapter. Netgear G622T Gigabit adapters connect the web booster and the web server. The MTU size for the link between the booster and the web server is set to 4KB using the jumbo frame capability of the Gigabit network adapters. The web server run the web accelerator and delivers all requested documents from packet cache, which has been warmed up by working set of resources before running the tests.

VII. RESULTS

This section studies the performance of the web booster under a workload that simulates network traffic for public web sites under WAN conditions. First, this section studies how the fraction of packets available for acknowledgment aggregation is affected by file size, MTU size of the client connections, and presence of the persistent connections. Next, it shows the effects of the described optimizations on the booster processing cost. The section also shows how the file size, the network parameters, and the presence of the persistent connections affect the booster processing overhead. Finally, it shows how the end-user request latency is affected by the network parameters of the client connection.

Figure 5 shows how the fraction of packets available for acknowledgment aggregation changes with the size of the document being transferred. This experiment uses the micro-benchmarking workload with requests to a single document with the size varied from 2KB to 60KB. The network delay is set at 200 ms, client connection bandwidth limit at 56kbps, and MTU size of the connections at 536B, which are typical parameters for public web sites. The graph shows the total number of packets received, the number of packets received from the clients, the number of packets received in FIN_WAIT_1 state and the number of packets that are available for the aggregation. Figure 6 shows the dependency of the fraction on the MTU size. This experiment used SURGE workload with the 10KB average object size, which leads to different number of packets received compared to the previous experiment. Both graphs show the number of packets normalized for the same client data sent rate of 6MB/s.

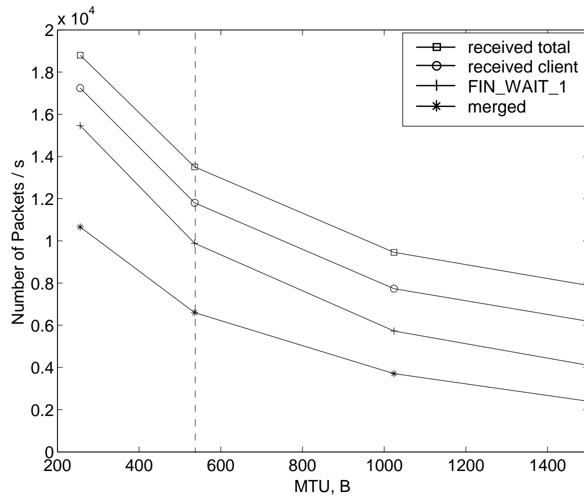

Figure 6. Number of packets vs. MTU

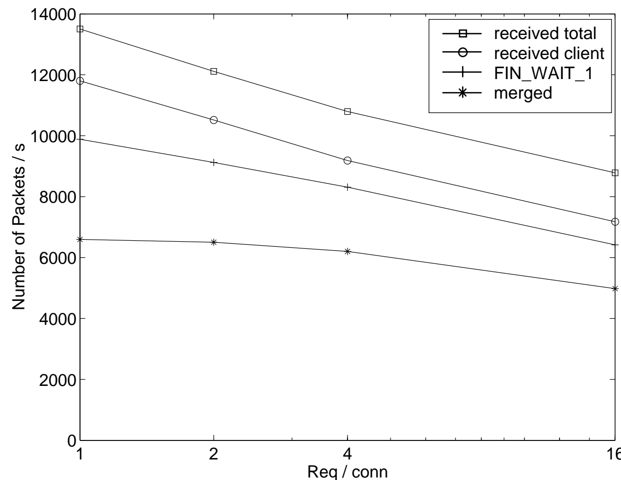

Figure 7. Number of packets for persistent connections

These two graphs show that the size of the objects has a significant effect on the acknowledgement aggregation availability. The aggregation technique becomes effective only for objects larger that 10KB in size. The effectiveness also decreases with the increase of MTU size. For the base case of 10KB document size, 536 bytes MTU size, 200 ms delay and 56 kbps bandwidth limit, about 50% packets on average are available for aggregation. The experiments also confirm the assumption that most of the received packets are processed in FIN_WAIT_1 state.

Figure 7 shows how the number of packets received in different states depends on the number of requests sent per persistent connection. The SURGE tool generates the workload with 10KB average object size for this experiment. Network parameters are set the same as for the previous two. The graph shows that even for the case of 16 requests per connection, the number of aggregated packets is still about 50%. Because the booster limits the client socket send buffer only by the total size of the data buffered at the booster, it can fit all 16 responses in the client socket and then close the connection switching it to FIN_WAIT_1 state. In the case if the socket write buffer had a limited size, as it is common for web

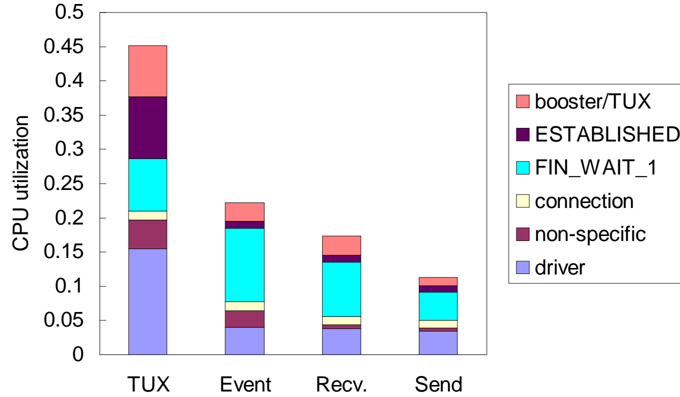

Figure 8. Booster cost breakdown comparison

servers, the fraction of packets available for aggregation would significantly dropped due to the small number of packets processed in FIN_WAIT_1 state. This makes acknowledgement aggregation effective only for large socket send buffers when persistent connections are used.

The next set of experiments studies the effect of the optimizations described in the previous section. Figure 8 compares request processing cost breakdown for three booster configurations and the TUX web server. TUX web server runs on top of Linux 2.4 and is implemented as a kernel daemon [37]. It is used as a base for the optimization comparisons because it uses the same TCP/IP implementation as the booster, without any optimizations incorporated (including device polling). The first booster configuration shown on Figure 8 has only event polling enabled, the second one adds two receive processing optimizations and the third one adds two send processing optimizations. The SURGE workload and the base network parameters were used for the experiments. The graph shows the following components.

- *Driver processing*. This includes removing received packets from the device hardware receive queue and passing them to the network protocol stack. If device polling is not used, as for TUX, the packets are queued at the protocol *backlog queue* instead, and the processing is performed in the hardware interrupt context. The hardware receive queue is refilled with newly allocated empty packets. The processing also includes freeing sent packets from the device hardware send queue.
- *State non-specific*. This item includes IP processing for the received packets, such as IP checksum verification and IP route table lookup. If the device polling is not used, the packets are removed from the backlog queue, and this and the rest of the items are processed in the software interrupt context. This item also includes the initial TCP packet processing before the state-specific processing path, such as destination socket lookup.
- *Connection open/close*. This includes processing specific to TCP connection opening and closing, excluding FIN_WAIT_1 processing.
- *TCP FIN_WAIT_1 state processing*. This includes processing in the FIN_WAIT_1 state and sending data as result of this processing. The cost excludes processing done in the booster module and the cost of sending data from the booster module.[1]

---

[1] Data sending may be performed either completely in the sender application context, in this case the booster module, or if data cannot be sent immediately, they are queued at the socket and sent later when the arriving

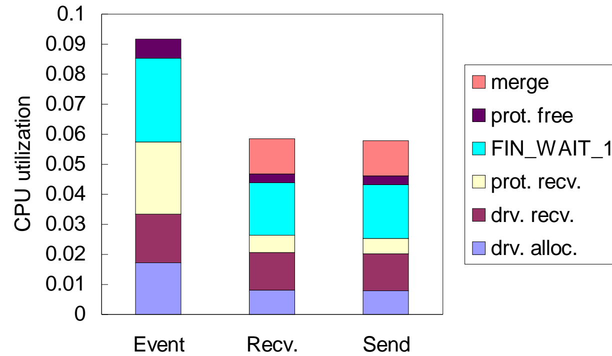

Figure 9. Receive cost breakdown

- *TCP ESTABLISHED state processing*. This includes processing in the ESTABLISHED state and the cost of sending packets as result of this processing. Similar to the previous item, it excludes the cost of processing in the booster module.
- *Booster/TUX processing*. This includes processing done in the booster/TUX module as well as processing for sending data from the booster/TUX module.

Figure 8 shows that the cost associated with processing in the ESTABLISHED state and with connection open/close is small compared to the other four components. The major contribution in the processing cost is processing in the FIN_WAIT_1 state. The contribution of the processing in the FIN_WAIT_1 state into the overall cost is even higher, if we take into account that most of the overhead of the driver and protocol state non-specific processing is due to the processing of packets that are further processed by FIN_WAIT_1 state specific path, as can be seen from Figure 6.

From Figure 8 we also can conclude that the cost associated with network processing of the server connections is low. Because during normal operation there is no connection open/close and all packets for server connections are processed in the ESTABLISHED state, this cost is part of the ESTABLISHED processing cost and a small fraction of the driver and state non-specific processing. The latter cost is proportional to the fraction of packets processed in the ESTABLISHED state. Both costs are an order of magnitude smaller than the total processing cost.

The cost of booster processing in the base configuration (only event polling) is half that of TUX processing. The main contribution in this decrease is driver processing. It is decreased by almost a factor of three due to the use of polling instead of the hardware interrupts and streamlining of the packet passing to the protocol level. Another contribution is the difference between the booster and the HTTP server cost. Figure 8 shows that TUX spends almost an equal amount of time in ESTABLISHED and FIN_WAIT_1 state processing while the booster processing in ESTABLISHED state is negligible. The four optimizations introduced in Section V together further decrease the processing cost by another factor of two to about one quarter that of TUX. We can see that the cost spent in FIN_WAIT_1 processing is significantly lower in the optimized configuration and now it is comparable to the cost of other components.

The next two graphs show a more detailed cost breakdown for the same experiments, which is subdivided into the two parts related to the receiving of the acknowledgments and to the sending of data

---

acknowledgments open the congestion window. In the latter case, the sending is accomplished in the same context as acknowledgment receiving and its cost is included in the described component.

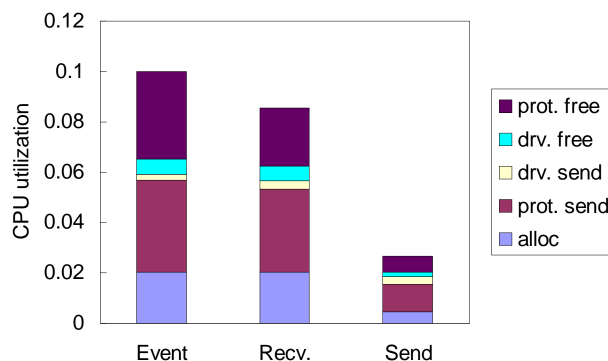

Figure 10. Send cost breakdown

caused by the acknowledgments. In these graphs we separately identified costs related to packet creation and freeing. The scale of the Y-axis on these graphs is different from the previous one. Processing related to acknowledgment receiving is shown on Figure 9 and includes the following.

- *Allocation of the packets in the device driver.* Some device drivers, including the one for the 3C590 card used for our implementation, copy small packets into newly allocated buffer of smaller size before passing them to the protocol stack to save the memory. Acknowledgment packets are of such a small size.
- *Driver receive operation.* This includes scanning through the adapter's receive buffers for the received packets and passing them to the protocol layer. As we mentioned, this operation may involve packet copying, but we included its cost in the previous item.
- *State non-specific receive processing.* This is similar to the one described in Figure 8 and includes protocol processing before entering a state-specific path.
- *FIN_WAIT_1 processing.* This processing excludes operations related to data sending, which are described later. This is different from the FIN_WAIT_1 component shown in Figure 8, where send processing cost incurred in network protocol stack was included.
- *Packet freeing.* This item includes de-allocation of the received packets by the TCP module.
- *Merge.* This is the cost of packet merging and the packet verification necessary for both the acknowledgment aggregation and the fast receive path.

Figure 9 shows that two receive processing optimizations significantly decrease the processing overhead associated with the state non-specific protocol processing. They also decrease the allocation cost in the device driver. The receive cost is half that of the basic configuration.

The processing related to the data sending is shown on Figure 10 and includes the following.

- *Packet allocation and queuing at the client socket write queue.* This operation is done in the booster module. The operation may also include sending of the first packet, which is allowed by the initial congestion window.
- *Network protocol part of the packet sending.* This operation is invoked when an acknowledgment arrives to verify that data can be sent at that time. It includes virtual copying or building of the packets to be sent and TCP/IP header construction. It excludes send processing in the device driver.
- *Driver part of the packet sending.* Invoked by the operations described in the previous item.

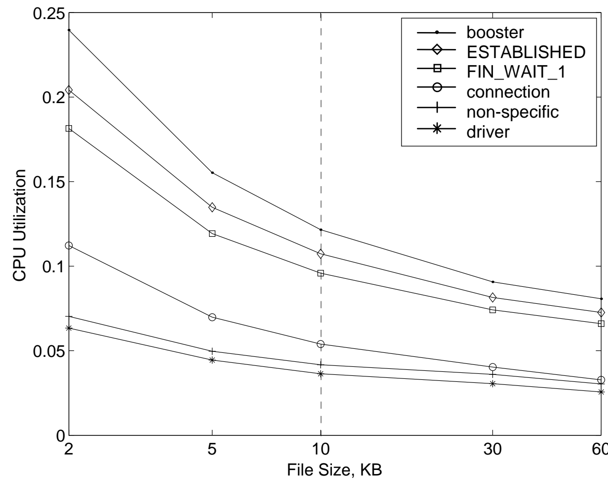

Figure 11. Cost breakdown vs. file size

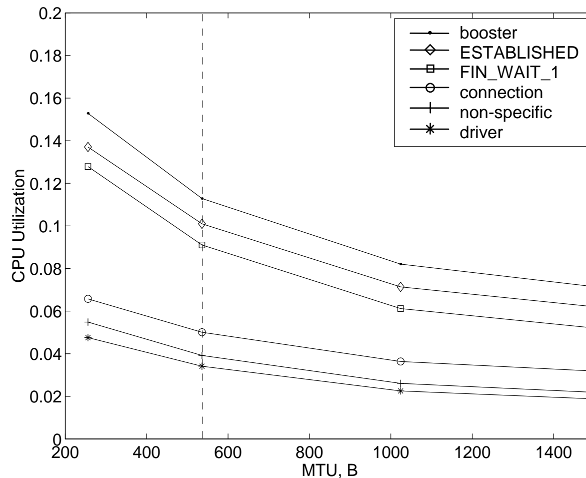

Figure 12. Cost breakdown vs. MTU

- *Packet freeing in the driver.* This is done in the device driver after the send operation completes. The virtual copy of the packet queued at the socket send queue or a new packet built in optimized TCP module is freed.
- *Packet freeing in the network protocol stack.* When the sent data are acknowledged, the original packets on the socket send queue are freed.

Two send path optimizations decrease the cost of packet sending by almost a factor of three compared to the base booster configuration. This is mainly due to the decrease of the protocol sending overhead and the decrease of the packet handling overhead at the socket send queue.

The next set of experiments studies the dependency of the web booster processing on the object size, usage of the HTTP/1.1 persistent connections, and network parameters of the client connections (Figure 11 - Figure 15). Those network parameters that are not varied in the experiments are set at the base values (200ms, 56kbps and 536B). The experiment with the object size uses micro-benchmarking tool for

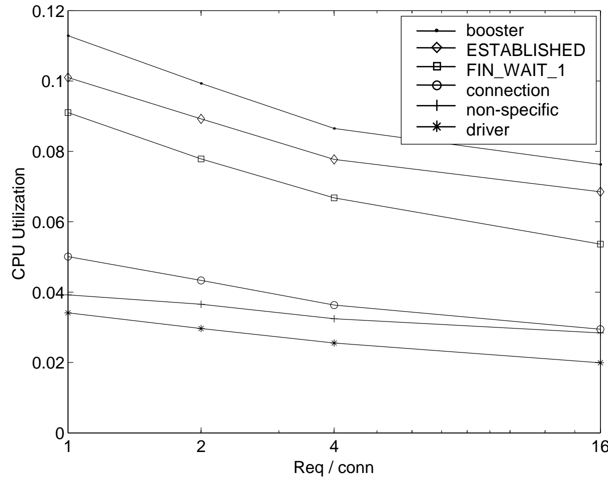

Figure 13. Cost breakdown for persistent connections

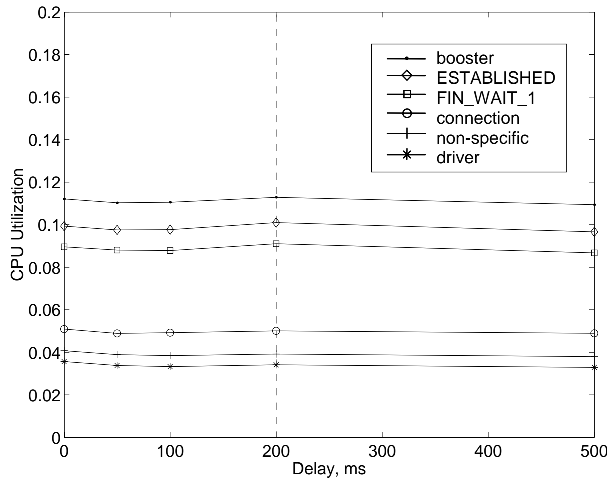

Figure 14. Cost breakdown vs. delay

workload generation and the other three experiments use the SURGE tool with 10KB average object size. The graphs show the same cost components as in Figure 8 as a cumulative stack, where each curve is the sum of the plotted component and the all components plotted bellow it. The CPU utilization is normalized to the 6MB/s client send data rate. The booster uses all optimizations described previously.

Figure 11 shows the cost breakdown dependency on the size of the responses sent to the clients. We can see that with the size increase, two components of the processing cost become dominant: processing in FIN_WAIT_1 state and the driver processing. This graph also shows that the overall processing overhead is significantly affected by the size of the responses mainly due to the connection open and close overhead, driver processing, and booster module processing.

Figure 12 shows the cost breakdown dependency on the MTU size of the client connections. An increase in the MTU size causes a decrease in the number of data packets sent and in the number of acknowledgements processed. This in turn decreases the cost of the FIN_WAIT_1 state processing and the driver processing, as can be seen from the graph. These two components decrease the overall

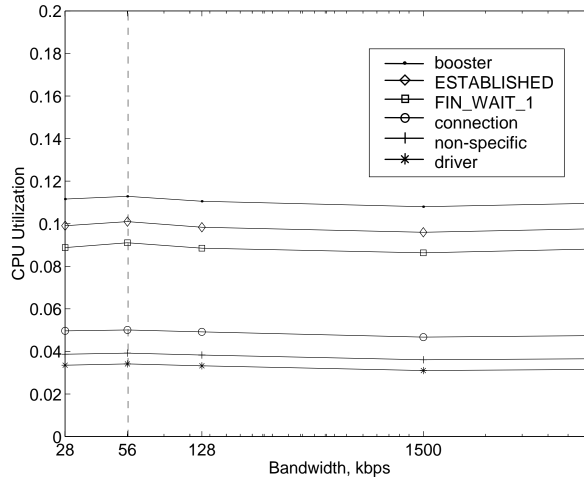

Figure 15. Cost breakdown vs. bandwidth

processing overhead more than by a factor of two when the MTU size is increased from 256 bytes to 1500 bytes.

Figure 13 shows effects of the HTTP/1.1 persistent connections on the processing overhead. Sending 16 requests per one connection decreases the overhead by about 20%. This decrease is mainly due to eliminating the cost of the connection open and close and decreasing the hardware interrupt overhead. Although the cost of the processing in FIN_WAIT_1 state is also decreasing, the combined cost of processing in FIN_WAIT_1 and ESTABLISHED states stays about the same.

The next two experiments vary network delay and bandwidth limit of the client connections. Although these parameters affect the latency of requests, their effect on the processing overhead is minimal, as Figure 14 and Figure 15 show.

From Figure 8 we can extrapolate the maximum performance of the web booster on our test platform. We could not actually measure the maximum performance due to the limits of the network links and the client machines that generated the workload. Figure 8 shows CPU utilization of the booster for 6MB/s send data rate, which is about 12%. For 100% CPU utilization, the extrapolation gives 50MB maximum data send rate, which is about 5000 requests per second for 10KB average object size. This is about 4 times greater than the performance of the TUX web server directly accepting client requests.

VIII. CONCLUSION

This paper described how knowledge about common case in network traffic can be applied to optimization of the general-purpose network protocol stack. Such optimizations leaded to four-fold increase in the throughput compared to the base case.

This study did not take into account packet losses and out-of-order packets in the network. Presence of such packets may change the fraction of packets available for aggregation. Seshan et al. [34] measured packet loss rate for the live web server traffic, which was in the order of 10%. We expect that such packet loss rate would not significantly change the effect of this optimization. The other three optimizations should not be affected by such packets.

The optimizations described can be applied to any TCP forwarding appliance that handles HTTP server traffic. In fact, the first two receive processing optimizations can be used for any network protocol stack that handles mainly unidirectional outbound TCP stream with mostly acknowledgement packets received. Particularly, these optimizations can be applied to the web server directly if the changes to the protocol

stack are acceptable. This is also true for the packet reuse optimization. Double allocation avoidance utilizes the fact that large MTU packets are received from the server connection and can be used directly on the client connection. In a context where there is no forwarding, for example in web server, this optimization cannot be directly applied. To adopt this optimization to the web server protocol stack, it would be necessary to queue the memory pages on the client socket directly and create packets only during the actual send operation. Implementing this optimization would require more substantial changes to the operating system and network protocol stack.

REFERENCES

[1] Alteon WebSystems. ACEdirector. http://www.nortel.com

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandervoorde, 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 (SOSP),* Saint-Malo, France, October 1997.

[3] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha. Design, Implementation and Performance of a Content-Based Switch. In *Proceedings of IEEE INFOCOM*, March 2000.

[4] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient Support for P-HTTP in Cluster-based Web Servers. In *Proceedings of the 1999 USENIX Annual Technical Conference, Monterey*, CA, June 1999.

[5] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 2000 Annual USENIX technical Conference*, San Diego, CA, June 2000.

[6] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[7] P. Barford and M. E. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of Performance '98/ACM SIGMETRICS '98*, Madison, WI, 1998.

[8] BoostWorks, inc. http://www.boostworks.com.

[9] J. Brustoloni. Exposed Buffering and Sub-Datagram Flow Control in ATM LANs. In *Proceedings of IEEE Local Computer Networks*, Oct. 1994.

[10] M. Carson. NIST Net. http://www.antd.nist.gov/nistnet.

[11] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *Proceedings of 2001 USENIX Annual Technical Conference*, June 2001.

[12] Cisco Local Director. Technical White Paper, 1998. Cisco Systems.

[13] A. Cohen, S. Rangarajan, and H. Slye. On the Performance of TCP Splicing for URL-Aware Redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct.1999.

[14] Y. Hu, A. Nanda, Q. Yang. Measurement, Analysis and Performance Improvements of Apache Web Server. Technical Report 1097-0001, University of Rhode Island, RI, Oct. 1997.

[15] IBM Corporation. IBM Interactive Network Dispatcher. http://www.ibm.com/software/network/dispatcher.

[16] V. Jacobson. 4BSD TCP Header Prediction. In *Computer Communication Review*, vol. 20, no. 2, April 1990.

[17] V. Jacobson. A High-Performance TCP/IP Implementation. Gigabit-per-Second TCP Workshop. CNRI, March 1993.

[18] P. E. McKenney and K. F. Dove. Efficient Demultiplexing of Incoming TCP Packets. In *Proceedings of the SIGCOMM ’92 Conference*, Aug. 1993.

[19] J. Liedtke, V. V. Panteleenko, T. Jaeger, and N. Islam. High-Performance Caching With the Lava Hit-Server. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[20] D. A. Maltz and P. Bhagwat. TCP splicing for application layer proxy performance. Technical Report RC21139, IBM Corporation, 1998.

[21] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the ACM SIGMETRICS '97 Conference*, Seattle, WA, June 1997.

[22] J. C. Mogul. Network Behavior of a Busy Web Server and its Clients. *Technical Report WRL 95/5*, DEC Western Research Laboratory, Palo Alto, CA, Oct. 1995.

[23] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Transactions on Computer Systems*, 15(3), Aug. 1997.

[24] D. Mosberger, L. L. Peterson, P. G. Bridges, S. O’Malley. Analysis of Techniques to Improve Protocol Processing Latency. In *Proceedings of ACM SIGCOM’96 Conference*, 1996.

[25] E. M. Nahum, M. C. Rosu, S. Seshan, and J. Almeida. The Effects of Wide-Area Conditions on WWW Server Performance. In *Proceeding of ACM SIGMETRICS’00 Conference*. 2000.

[26] NetScaler. http://www.netscaler.com.
[27] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.
[28] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Sever. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.
[29] V. V. Panteleenko and V. W. Freeh. Web Server Performance in a WAN Environment. Technical Report TR-02-02, University of Notre Dame, July 2002.
[30] V. V. Panteleenko and V. W. Freeh. Web Booster Architecture for Accelerating Web Servers. Technical Report TR-02-04, University of Notre Dame, July 2002.
[31] C. Partridge and S. Pink. A Faster UDP. In *IEEE/ACM Transactions on Networking*, vol.1, no. 4, Aug. 1993.
[32] Redline Networks. http://www.redlinenetworks.com.
[33] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. In *Computer Communication Revue*, 27(2), Feb. 1997.
[34] S. Seshan, H. Balakrishnan, V. N. Padmanabhan, M. Stemm, and R. H. Katz. TCP Behavior of a Busy Internet Server: Analysis and Improvements. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM) '98*, San Francisco, CA, March 1998.
[35] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing TCP Forwarder Performance. Technical Report TR98-01, University of Arizona, Feb. 1998.
[36] The Standard Performance Evaluation Corporation. SPECWeb96/SPECWeb99. http://www.spec.org/osg/.
[37] TUX Web Server 2.0. http://www.redhat.com/products/software/tux.
[38] C.-S. Yang and M.-Y. Luo. Efficient Support for Content-Based Routing in Web Server Clusters. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.