

Web Booster Architecture for Accelerating Web Servers

Vsevolod V. Panteleenko and Vincent W. Freeh

TR-02-04

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

Abstract—This paper describes a novel approach to increase the performance of web sites by changing the network traffic between the client and the web servers. This is achieved by introducing a web booster architecture, which instantaneously decreases the processing cost on a web server for requests to both static and dynamic web resources. It consists of a booster appliance, which pre-processes client requests, and an accelerator software module running on the web server. Fast reaction to the load change and the release of resources during inactive period enables multiple web servers to efficiently time share the booster appliance.

The paper describes six techniques that decrease cost of the server processing up to nine times. These techniques are: (i) increasing MTU size of the connections to a web server, (ii) avoiding connection open and close, (iii) delaying client acknowledgements, (iv) performing request processing in the kernel, (v) avoiding data copying and checksumming, and (vi) avoiding packet creation overhead. The paper describes a web accelerator software module running on the web server that enables the last three of the above optimizations by caching the requested documents in the kernel in the form of TCP packets. The paper also presents a design of the booster appliance and four novel optimizations of its network protocol stack that increase booster performance up to a factor of four.

I. INTRODUCTION

In order to increase the performance of web sites, designers have traditionally used several approaches: increasing single web server performance, adding more web servers in parallel, and placing caches in front of the web servers. This paper introduces a novel approach for increasing web site performance: changing network traffic between clients and the web server in order to reduce the cost of request processing on a web server. This is a grey box approach [7], where changing the use of external interfaces

of the server (in this case changing incoming network traffic) modifies its internal behavior. It requires only general knowledge of server internal structure and does not require any changes to the internal implementation of the server. Network traffic is modified by placing a *web booster* appliance in front of the web servers. The “cooked” request traffic from the booster is about nine times easier for the server to “digest” than “raw” traffic from clients.

A web booster can *instantaneously* decrease the overhead of request processing on a web server. Therefore, a web booster can be used as a shared resource for a set of web servers, providing on-demand offloading of a currently overloaded server. This property distinguishes the booster from a web cache, which requires a warm-up period before it can effectively serve a new server, or it must keep a working set of documents for all the web servers. A web booster leaves all request processing at the web servers, which makes it transparent and readily deployable. In addition, unlike a cache a web booster decreases the processing overhead for both static and dynamic documents.

The web booster modifies client request traffic in three ways. First, it changes MTU size of the server network interface into the largest MTU size that can be supported on a link between the booster and the server. This greatly decreases the cost of packet processing on the server. Second, the requests are sent through several persistent TCP connections. This eliminates connection open and close overhead and throughput delays due to TCP slow start. Third, the booster delays acknowledgements beyond the TCP standard. This also reduces the overhead of packet processing on the server. Although it is technically a violation of TCP, it is implemented on internal LAN connection between the booster and the server and does not require a change to the TCP module on the server. This paper presents a detailed study that shows how each of the above techniques affects server processing.

This paper also presents an implementation of the web booster architecture on top of Linux for Intel Pentium-class servers. There are two components to the implementation. A web booster appliance that operates as a standalone, front-end machine and an *accelerator* kernel module that runs on web servers. The latter removes overhead of the data copying in the network protocol stack on servers that do not support zero-copy framework.

The prototype booster appliance uses four novel optimization of the TCP/IP stack processing. This enables the booster to forward requests at a rate four times greater than the web server process them on the same hardware platform. Thus, a network of two boosters and a single server can handle eight times the maximum traffic of the original server. This is nearly three times better than using the same machines as three independent parallel servers.

This paper is organized as follows. The next section describes related work. Section III presents the booster architecture design. Section VI measures performance of a general-purpose web server that processes requests directly from the clients. The results of this section are used to design acceleration techniques for a web server, as described in Section V. Section VI gives an overview of the booster appliance design. The performance measurements of the booster architecture are given in Section VII. Finally, Section VIII concludes.

II. RELATED WORK

The most widely used approach to offload web servers is using reverse proxy caches that are located in front of a web site. There is a large body of work that studies different aspects of web proxy caching such as web traffic characteristics from the viewpoint of proxy caches [3][15][16][23] and cooperative caching [19][29][47][50]. These studies show that the effectiveness of caching is limited by factors such as resource changes and the number of non-cacheable resources. For a forward proxy cache, the hit rate is not larger than 40-60%, even for caches serving very large populations.

A web server can be replicated to handle more requests. To distribute requests to different web servers, front-end appliances accept client requests and forward them to the servers based on some policy. One example of a distribution policy is weighted round-robin distribution, which is used in many commercial

appliances [4][21][26]. Request distribution requires TCP forwarding support in the appliance, which forwards packets based on the TCP connection. 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 [8][37] requires layer-7 (application) forwarding support in the forwarding appliance.

A web site might have other types of front-end appliances besides request distributors. Such appliances may offload computationally expensive tasks from the web servers. An example is the SSL accelerator [5] that offloads secure socket layer (SSL) encryption and decryption processing. There are several commercial front-end appliances that decrease the cost of network processing on web servers [14][36][42]. Their approach appears to be similar to the one proposed in this paper. Unfortunately, there are no details of the design and implementation for these appliances publicly available.

Several existing systems use the Internet infrastructure to temporarily or permanently offload web sites. Two commercial systems, Akamai [1] and Digital Island [22], provide a set of high-performance web servers distributed throughout the Internet. Web sites that subscribe to this service upload large web resources to these servers, which, in turn, deliver them to clients. Web servers still accept client requests for the majority of resources but the resource transfers with high processing and network requirements are performed by the dedicated servers provided by these companies. In this case, the web providers retain control of their web sites without owning or maintaining high-performance hardware.

There is large body of work that studies the performance of web servers. Such studies use either real-life workload [31][33][44] or simulated workload [10][13][25][35]. Most of these studies show that the network processing on web servers has a major contribution to the overall processing cost. These studies also show that failing to reproduce wide area network (WAN) conditions for the simulated workload can lead to significant differences in web server behavior compared to real-life workload.

Many projects have attempted to optimize web server performance by changing the design of the HTTP servers and modifying operating system and network protocol processing. Pai et al. looked into the performance of different server architectures in the context of one implementation: the Flash web server [39]. The study shows that combining event-based processing for requests that hit the memory cache and process-based handling of the file system accesses performs better than other types of architectures.

To decrease the overhead of context switches, HTTP servers can be implemented as kernel daemons [28][32][48][49]. This design may also allow tighter integration of the server with the operating system and the network protocol stack. For example, a kernel HTTP server may directly access the file cache, which may not be possible from the user level. A different approach for tighter integration with the operating system is to provide specialized system calls that are designed with web servers in mind [11]. The Lava caching server shows the effect of using a specialized operating system and building the server as integrated software [30]. This design shows almost an order of magnitude better performance than other research web servers on the same hardware.

Several studies identified data copying and checksumming in the network protocol stack as the significant overhead for web servers. BSD and Linux 2.4 kernels introduce a zero-copy framework, where data copying and checksumming during socket send operation are avoided by using scatter/gather and hardware checksumming capabilities of a network adapter. IO-Lite goes further by introducing a unified I/O buffering and caching system [38].

To alleviate the overhead of event dispatching in web servers, such as *select* system call overhead, the implementation of this system call was modified to avoid unnecessary scanning of the file descriptors that are known to be not ready at the time of the scan [12]. Another approach is to use a different event dispatching mechanism. Chandra et al. [18] suggested using POSIX.4 Real Time signals, which are more efficient than *select* or */dev/pool*. This study claims that even an inefficient *select* has low overhead for a loaded web server because its cost is amortized over many requests. This claim is supported by a study in this paper.

Another optimization of the event dispatching is avoiding hardware and software interrupts. Several studies [20][34] suggest using device driver polling by periodically checking the network adapter for arrived or sent packets. In this case, all network processing is performed in a process context instead of interrupt context. This optimization is applicable only for specialized environments that have high bulk transfer rate, such as a busy web server. For interactive applications, this optimization introduces high latency.

The APFA platform [28] attempts to decrease the overhead of event dispatching even further by performing HTTP request processing in the socket event handler in software interrupt context. This platform also implements many of the optimizations described in this section, including data copy/checksum avoidance and minimizing context switches by processing in the kernel.

An even more dramatic change to the network protocol processing is offloading computationally intensive operations to a network adapter with on-board processors. The Alachritech [2] network adapter is capable of performing packet fragmentation and de-fragmentation, TCP acknowledgement processing, partial TCP connection management, and data copying directly to user-level buffers as well as performing more standard operations, such as checksum computation and interrupt coalescing.

Another approach that removes the overhead of network protocol processing from a web server is migrating a communication link between the web server and the front-end appliance of a web site from the existing LAN technologies, such as Ethernet, to a System Area Network (SAN) architecture [24][27][45]. This architecture provides reliable communication in hardware and uses a computationally light-weight transport protocol, which is different from TCP. This approach requires the front-end appliance to accept client TCP connections and forward data to and from the web server using the SAN transport protocol.

This paper takes a conservative approach for web server performance improvements. It does not change either the operating system, the protocol stack of a web server, or the underlying network technology. This retains the existing web site infrastructure and legacy hardware and software.

III. WEB BOOSTER ARCHITECTURE

Public web sites are usually built in a multi-tier architecture. First-tier web servers handle requests for static resources redirecting other types of requests to second-tier application servers and third-tier database servers. A web site usually has additional appliances, such as web caches and load balancers, which are located in front of the first-tier servers. Many public web sites host multiple administrative domains, which might use dedicated servers for quality-of-service guarantee and protection or share servers with each other for better hardware utilization.

Web booster architecture enables offloading of the network protocol processing at the first-tier web servers. This offloading is accomplished by changing network traffic to and from the web servers by a *web booster appliance* located in front of them (see Figure 1). Traffic is changed at the network and upper protocol layers. Such changes decrease the processing required of the server's protocol stack, migrating this processing to the booster. Unlike with other mechanisms, the overhead decreases for both static and dynamic resource processing. As can be seen from the test results presented in the next section, network protocol processing is the main contribution to the overall web server load at public web sites. This mechanism does not require changes to the web server configuration and can work with any type of hardware and software. In some cases though, particularly if the server operating system or network hardware does not support zero-copy data sending, the web booster architecture adds to a web server a software module called *web accelerator*, which eliminates copying and packet creation overhead by providing a packet cache.

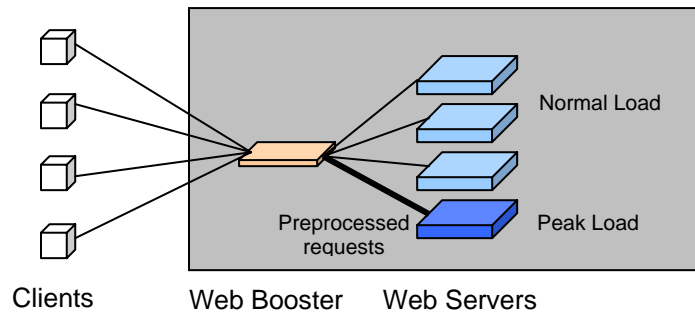


Figure 1. Web booster architecture

Offloading of a web server using this technique has two important properties. First, offloading is fast because there is no state to migrate. This enables migrating processing between an overloaded server and a booster on-demand. This property can be useful in the case when a web site hosts multiple domains that have dedicated web servers. If such servers have low correlation in traffic, excessive processing required during peak load at a server may be efficiently migrated to the shared web booster, decreasing hardware requirements for the server and thus for the whole web site.

Such on-demand sharing cannot be efficiently performed by a traditional front-end caching proxy. A caching proxy requires sufficiently long “warm-up” period to acquire the necessary resources for a high hit rate, which makes switching between servers an expensive operation. In contrast, a web booster can instantaneously switch from one server to another following the distribution of the server traffic. A web booster has an additional advantage over a web cache by being insensitive to the working set size of the server’s resources or to a fraction of the cacheable resources. A web cache has to provide sufficiently large main memory to accommodate the working set, and the latter has to have a significant fraction of cacheable resources, to efficiently offload the web server.

Second, migrated network processing can be potentially done more efficiently on a web booster than on a web server. A web booster is a specialized appliance that can be tailored to perform network processing for web traffic, possibly even utilizing specialized hardware. In this case, a web booster can be used in front of both shared and dedicated web servers to increase their performance and/or to lower total hardware requirements. This is especially important if web servers run a general-purpose operating system and modifications to servers’ configurations are not acceptable. In addition to this, a web booster leaves all HTTP-level request processing semantics at the first-tier servers, including such features as request logging and resource invalidation policies.

Section VI describes the design of a booster appliance that is built using off-the-shelf hardware (PC) and a general-purpose operating system (Linux). The latter was modified to introduce specialized network processing required to increase booster performance. The booster can process four times as many requests as a general purpose web server on the same hardware platform and operating system (unmodified). At the same time, a web server can process nearly an order of magnitude more requests when the network protocol processing is offloaded to the booster(s). This makes it possible to use two web boosters and one web server with the equivalent capacity of eight stand-alone web servers, assuming that all machines are of the same platform.

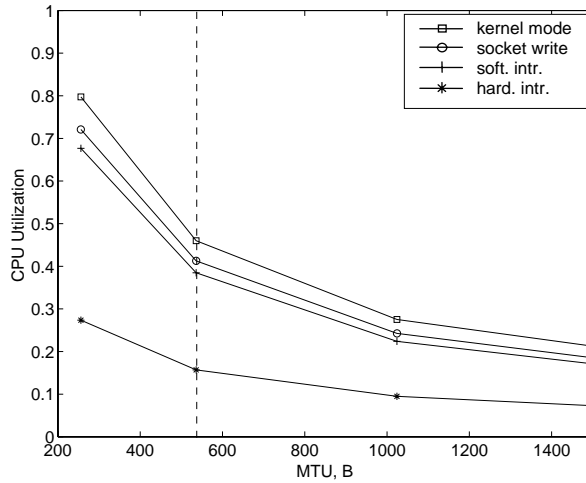


Figure 2. TUX cost breakdown vs. MTU

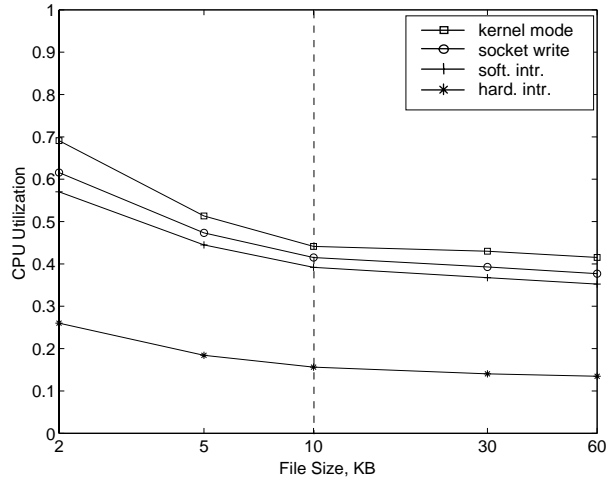


Figure 3. TUX cost breakdown vs. file size

IV. WEB SERVER REQUEST PROCESSING

This section evaluates the performance of a web server under a workload that emulates wide area network (WAN) environment. This identifies the overhead associated with different operations and suggests ways to decrease the overhead. The measurements are performed with the TUX HTTP server [48], which is implemented as a kernel daemon. The experiments use Intel Pentium III 650MHz server machine that runs Linux 2.4 kernel. The server workload emulates wide range of real-life workload characteristics for public web servers, particularly network delay, limited bandwidth of client connections, and small MTU sizes. The workload characteristics are similar to those used for the booster experiment described in Section VII. All requests are delivered from the memory cache of the web server. Detailed description of this study can be found in Panteleenko and Freeh [40].

Figure 2 and Figure 3 show the breakdown of request processing costs into several components: the fraction of time spent in hardware interrupts, in software interrupts, in process context in kernel mode, in

user mode and, as a separate item, the time spent in the socket *write* system call in the process context. The last component is a part of the kernel-level processing in process context, but is graphed separately. These graphs show a cumulative stack of the components, where each curve is the sum of the plotted component and the all components plotted below it. Because TUX is implemented as a kernel daemon, the time spent in kernel mode in process context includes HTTP related processing.

Figure 2 shows the breakdown dependency on the MTU size of the incoming client connections. For these experiments, the SURGE workload tool, described in Section VII, generates requests with an average object size of 10 KB and MTU size ranging from 256B to 1500B. Network parameters are set at 200 ms for the network delay and 56 kbps for the connection bandwidth limit. Figure 3 shows such overhead dependency on the file size under micro-benchmark workload, which generates requests for a single document. This experiment varies file size from 2KB to 60 KB and uses the same network parameters as above with MTU size of 536B. The data send rate was kept at 6MB/s for both experiments.

We can see that in spite of the fact that most of the server data are sent rather than received, most of the time is spent in hardware and software interrupt handling, which is triggered by packet reception. This overhead is proportional to the number of packets processed. The time spent in the process context including HTTP processing is small, which indicates that most of the overhead for a server is due to the network protocol processing. The graphs show that this overhead is highly dependent on the MTU size of the client connections and less dependent on the file size, provided that the total data send rate is kept the same for different file sizes.

These experiments show that improving network protocol processing may introduce a significant decrease in overall request processing overhead. This is especially important for public web servers that have large fraction of dial-up users with small MTU sizes. This observation is the basis for the web server acceleration techniques introduced in this paper.

V. ACCELERATION TECHNIQUES

This section introduces six web server acceleration techniques that are used to offload a web server. They can be subdivided into two categories. The first category includes optimizations of the client request stream to decrease the cost of processing in the server's TCP/IP stack and device drivers. Such optimizations do not require any changes to the web server set-up. The second category optimizes request processing above the server's TCP/IP protocol stack. These optimizations require specialized software, called a web accelerator, that is located on the web server between the TCP/IP stack and the general purpose HTTP server. Although such software is platform dependent, it does not require the operating system, the device driver, or the HTTP server to support any special features; therefore, it can be readily ported to different platforms.

First, this section describes our acceleration techniques. Then, it introduces the design of the web accelerator. Finally, it shows the test results that substantiate that the described acceleration techniques improve the performance of a web server by up to an order of magnitude.

A. Network Processing Optimizations

The first category of the acceleration techniques optimizes network protocol processing at the web server. Such acceleration is achieved by decreasing the number of packets processed by the server. The previous section shows that this optimization significantly decreases the processing overhead. Decreasing the number of packets processed is accomplished by the following techniques:

- Increasing the MTU of the incoming packets,
- Avoiding TCP connection open and close, and
- Delaying TCP acknowledgements to the web server.

The number of packets processed by the web server depends on the MTU of the connections to the clients. The MTU for the connection is the smallest MTU on the path between the server and the client.

For the large portion of clients of public web sites, it becomes the MTU of the dial-up link, which is several times smaller than the MTU of a high-bandwidth LAN link.

The web booster terminates client TCP connections and connects to a web server using the large MTU size of the LAN connection. The actual MTU size depends on the network hardware used for the link and may be an order of magnitude larger than 536 bytes, which is common for dial-up connections. As follows from the data in the previous chapter, an increase in the MTU size from 536 bytes to 1500 bytes cuts network processing overhead by more than half. The effect should be even larger if jumbo Ethernet frames, with 4000 or 9600 byte MTUs, are used.

Connection open/close management starts to introduce relatively large overhead, when the number of packets sent and received by the web sever is significantly decreased by increasing MTU size. Therefore, the web booster opens a small number of permanent TCP connections to the web server and sends all the requests through these connections. This decreases the processing overhead in several ways. First, avoiding TCP SYN and FIN packets decreases the number of packets processed by the server. This also avoids expensive code paths in the TCP module specific to the connection open/close. Second, the HTTP server does not have to manage multiple connections, which may decrease the cost associated with the HTTP server. Third, the number of acknowledgments processed by the web server using permanently open connections is almost half of that when a separate connection is used for every request. This is because in the latter case, the connections are in TCP slow start phase during most of their lifetime. During slow start the peer socket sends an acknowledgement for every received packet rather than for every two packets as in regular mode.

As we saw in the previous section, acknowledgment processing introduces significant overhead. When the web booster and the web server are connected in a local area network, the rate of packet loss is low. In this case, it is possible to decrease the number of acknowledgement packets from the booster. The booster send acknowledgements less frequently than for every two packets received from the server, as the TCP protocol specifies. This technique requires a modification of the TCP module of the web booster but not the web server.

B. Web Accelerator

The second category of web server acceleration techniques deals with request processing above the TCP/IP stack. Unlike network processing optimizations, this category of optimizations is applicable only for static documents. To enable these techniques, a web accelerator has to be located in the kernel of a web server. The web accelerator decreases request processing cost by:

- Performing HTTP request processing for static documents completely in the kernel mode,
- Avoiding copying and checksumming for sent data, and
- 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. The web accelerator accepts HTTP requests from the booster or directly from the clients. It processes requests for static documents and redirects requests for dynamic documents to a general-purpose HTTP server running at the user level.

Figure 4 shows the design of the accelerator. It consists of two main parts: the accelerator core, which handles the request processing, and the packet cache. The accelerator uses a regular socket read to receive requests from the web booster or directly from the clients. If the request misses the packet cache, the accelerator reads files directly from the file system and builds the network packets that it stores in the packet cache. Partial checksums on the packet data are pre-computed and stored with the packet. After the packets are built or found in the cache, the accelerator uses a special entry point into the TCP module to send or queue the packets to the socket. This entry point avoids the regular packet creation mechanism and is provided by many TCP implementations, including Linux and BSD.

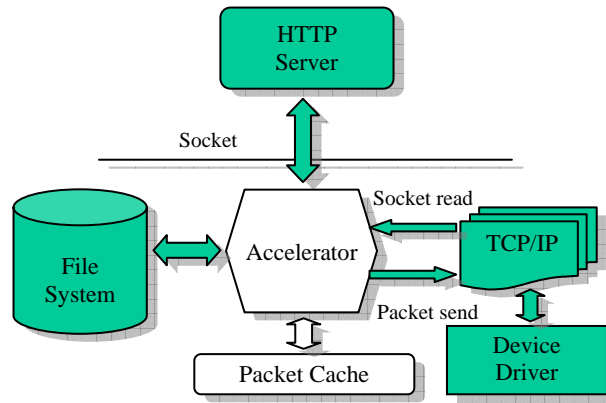


Figure 4. Accelerator design

If the request has to be delivered to the user-level HTTP server, the accelerator writes an unmodified request to a separate socket that is used by a user-level HTTP server to read requests. In this way, the user-level HTTP server processes requests that cannot be handled by the accelerator. These are requests to dynamic resources, the HTTP requests requiring modification of the object data, GET range requests, requests requiring special logging, etc. If the packet cache contains an object with the same URI, this object is invalidated (except GET range requests).

The packet caching technique can be used only within a range of the MTUs that are used for connection to the web booster or clients. If a packet for a requested document is larger than the MTU of the connection, it has to be fragmented, which involves copying and checksumming. If it is smaller, the number of processed network packets and the corresponding cost of request processing unnecessarily increases. When the web accelerator communicates with the web booster, the packet cache can be built for the larger MTU of the booster connections. If the accelerator receives requests directly from clients, it deals with a wide range of MTUs, which makes the packet cache significantly less efficient.

The accelerator requires the following support from the operating system and the network protocol stack.

- *Memory management.* Accelerator requires non-paged (pinned) memory allocation that is possible to use as network buffers.
- *Packet queuing.* The TCP implementation must export an entry point that allows sending or queuing network packets, thus avoiding the regular data copying and packet creation mechanism.
- *File system access.* Although it is possible to use the regular file read operation exported to user-level processes, it is more efficient to read file data directly into the accelerator memory, avoiding file buffering.

Most modern operating systems provide such support for the kernel modules, including Linux 2.4 kernel, which was used as our implementation platform.

The web accelerator was designed with the intention of accommodating a wide variety of operating systems and network hardware. Some of the optimizations described here, particularly copy/checksumming avoidance, are possible to implement without the packet cache if the network protocols stack and the network adapter support scatter/gather buffers and checksumming in hardware. In this case, running the web accelerator on the web server is not necessary for those acceleration techniques. Nevertheless, the web accelerator design is applicable in the cases where such hardware or software support is not available.

VI. WEB BOOSTER APPLIANCE

This section describes design of the web booster appliance that sits between clients and a server. Although the functionality of the booster is similar to web proxy functionality and it can be implemented at the application level on top of general-purpose operating system, a booster may easily become a bottleneck for the web site. This section outlines network protocol and device driver optimizations that improve booster performance up to four times compared to a straightforward implementation. In depth description of these optimizations and their performance analysis can be found in Panteleenko and Freeh [41].

The web booster appliance is a TCP forwarding appliance that terminates client TCP connections. It forwards client data to new connections opened 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.

The high-level HTTP booster module that runs on top of network protocol stack is responsible for handling HTTP-level semantics. It maintains a pool of permanently opened server TCP connections that multiplex client requests. The HTTP module accepts client requests, modifies HTTP header to set HTTP persistent connection option, finds a connection with lowest number of outstanding requests, and sends the request on this connection queuing the internal request handle to identify the response later. The module concurrently reads the responses from the server, identifies corresponding request handles and the client connections, modifies the headers to correspond to the original request, and sends the response to the clients. This module is implemented in the kernel and is integrated with the network protocol stack and the driver for performance reasons.

The operating system and the network device driver are changed to use polling of the device driver instead of using hardware and software interrupts [20][34]. 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 layer without queuing them at the general backlog queue. In the unmodified device layer, the device driver accomplishes such queuing in hardware interrupt context, and the queued packets are further processed in software interrupts.

Similar to a web server, most of the processing overhead for the web booster is due to network protocol stack processing. The booster design introduces four modifications to the network protocol stack and the network device driver that increase performance. These modifications decrease the number of fully processed network packets and the packet processing cost. These optimizations are: (i) incoming acknowledgement aggregation, (ii) fast path for incoming packets, (iii) double allocation avoidance in TCP module, and (iv) packet reuse. The first two optimizations are for the receive path in network protocol stack, and the last two are for the send path.

Acknowledgement aggregation combines several client acknowledgement packets into one packet early in the processing (in device driver). Properties of the booster network traffic make it possible in most cases to update the previous packet's acknowledgement sequence number and possibly the advertised window with values from a newly arrived packet. This decreases the number of fully processed packets and, in addition, saves the packet allocation and freeing cost for aggregated packets because the packet buffers are reused immediately. Measurements under simulated WAN workload show that for 536B MTU size and average 10KB file size, about half of the all received packets can be aggregated [41].

Fast path for incoming packets is an extension to acknowledgement aggregation. Most of the checks that a protocol stack performs in the device driver, incoming IP processing, and initial TCP processing are

also performed during acknowledgement aggregation. The fast path of the introduced optimization delivers packets directly to the TCP state-specific processing after checking for possible aggregation. This avoids above checks in the protocol stack.

The third optimization avoids double packet allocation during sending data to the clients. In the unmodified stack, packets received from the server are used to create client packets, usually of a smaller size. These client packets are allocated and queued at the client socket. During packet sending, a new virtual copy of the client packet is made by allocating a packet header and referencing the same data. The copied packet is passed to a device driver, and later freed by the driver. The packet queued at the socket is freed only when it is acknowledged. The introduced optimization queues *server* packets at the client socket rather than allocating new client packets. A new client packet is allocated only when it is passed to the device driver. Our implementation uses a zero-copy framework to create client packets. Scatter/gather packet header of a new packet points to multiple buffers in original server packets queued at the client socket. 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 at the client socket 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.

The fourth optimization reuses sent packets. For the booster network traffic, most of the sent packets already have almost all necessary information for subsequent packets for the same socket. These sent packets are queued at the special socket queue instead of being freed. During the next send operation, packet header information is updated with the new buffers and TCP sequence numbers and the packets are passed to the device driver. This operation saves one allocation and freeing operation and decreases the cost of send processing by avoiding the send processing pass in the network protocol stack.

The web booster was implemented using a general-purpose operating system (Linux 2.4). These optimizations require changes to the operating system, network protocol stack, and a device driver, as well as integration of the HTTP module with the network protocol stack. Some of these optimizations can be applied directly to the web server if the changes to the operating system and the device driver are acceptable, while others (double allocation avoidance) can be used only for the web booster.

VII. RESULTS

This section measures the effect of web server acceleration techniques using our implementation of the booster architecture. First, it describes the experimental methodology used. Next, it compares the performance of the accelerator module to the performance of the TUX server when both accept requests directly from the clients. Later, the section presents the breakdown of the processing overhead for the accelerator for the base case of no optimizations enabled followed by four measurements when each optimization is enabled one by one. It also shows the effect of interrupt coalescing for the accelerator when all optimizations are enabled. Finally, the section evaluates the performance of the web booster implementation and measures end user latency and latency of the booster activation.

A. Experimental Methodology

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

The second tool is used for generating a realistic client workload. It was built from the SURGE workload generation tool [13]. 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 [46], particularly self-similarity of the server network traffic produced by the requests [13].

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 [10] 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 [35] that used the Dummynet tool [43] 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 [17] 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.

Our fourth tool performs web server and web booster profiling. It measures the 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 [6]. 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 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.

All experiments in this paper study web server performance for memory-cached objects by warming up the file or packet cache with the objects before running the tests. The SURGE tool is configured with 146MB total size of the object set. The memory size of the test machines allows fitting the whole object set into the memory and making no file system accesses. These experiments cover the common case for commercial web sites, which install large main memory on front-end web servers to avoid file system accesses for the working set of objects.

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 the Linux 2.4 kernel. Each client was 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 connected the web booster and the web server. The MTU size for the link between the booster and the web server was set to 4KB using the jumbo frame capability of the Gigabit network adapters.

B. Base case

The first set of experiments measure the performance of the web accelerator when it accepts requests directly from the clients. The workload simulates WAN environment. The web booster is not used and no network processing optimizations are enabled in the experiments. The experiments provide the baseline for the further processing optimizations. They also give the performance comparison of the web accelerator with the other HTTP servers.

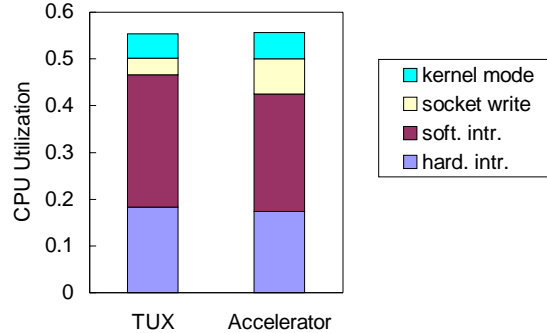


Figure 5. Comparison of accelerator and TUX

Figure 5 shows the breakdown of the processing cost for the web accelerator compared to the same breakdown for the TUX server. The graph shows a cumulative stack of the same components as graphs in Section IV: the fraction of time spent in hardware interrupts, in software interrupts, in process context in kernel mode, in user mode, and the time spent in the socket *write* system call in the process context. These experiments do not use the packet cache for copy, checksum, and packet management avoidance: data are copied in packets and checksummed on every send operation. The SURGE tool generates the workload for the experiments with 10KB average object size. The MTU size is set at 536 bytes, the network delay at 200 ms and the connection bandwidth limit at 56 kbps. As previously, the send data rate is 6MB/s.

The cost breakdown is similar for both servers and they have approximately the same CPU utilization. The accelerator has a higher cost for socket write because it copies and checksums the sent data, while TUX uses zero-copy socket write. Similar to the experiments in the Section IV, most of the overhead for the accelerator is due to the network processing in software and hardware interrupts.

C. Acceleration Techniques

The next set of experiments measures the effect of the optimization techniques introduced in Section 4. Experiments study the dependency of such optimizations on the requested object size. Workload is generated by the micro-benchmarking tool making requests to a single object with the object size varied from 2KB to 60KB. The following graphs plot the CPU utilization scaled to keep the same send data rate of 6MB/s for all graphs. As with the graphs in the Section IV, the following graphs show the cumulative stack of the components, where each curve is the sum of the plotted component and all components below it.

There are four optimizations that are enabled in sequence: connection MTU size is increased to the LAN size, connection open and close is eliminated using HTTP/1.1 persistent connections, copy and checksumming is avoided by using the accelerator packet cache, and acknowledgements from the booster are delayed beyond one acknowledgement for every two packets received. The sequence of these optimizations was intentionally chosen to go from the most significant to the least significant one. Each optimization decreases the overhead that becomes dominant after the previous optimizations.

Figure 6 shows the cost breakdown for the base case of no optimizations. Network parameters were set at 536 bytes MTU size, 56 kbps connection bandwidth limit and 200 ms network delay. The cost distribution for 10KB object size is different from the Figure 5 because of a different workload. CPU utilization of the web accelerator decreases when the object size increases until the size reaches 10 KB, after that it becomes almost flat. This decrease is mainly due to the hardware and software interrupt components.

Figure 7 shows the cost breakdown for 1500B MTU size. The scale of the Y-axis is half that of the previous graph. Compared to Figure 6, the graph shows the effect of increasing the MTU size to a LAN

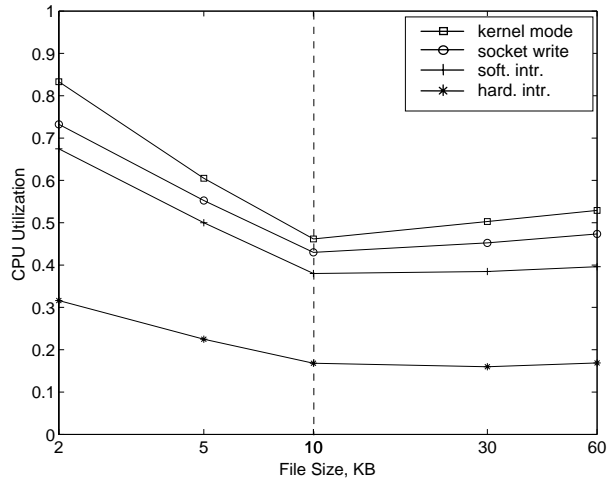


Figure 6. Cost breakdown vs. file size: base case

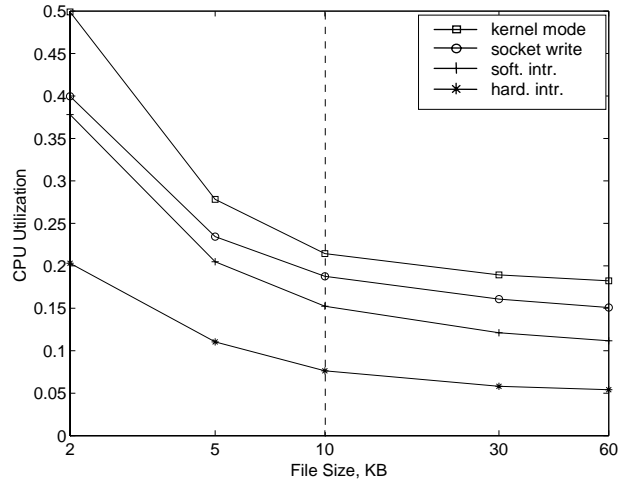


Figure 7. Cost breakdown vs. file size: large MTU

value and removing the network delay and bandwidth limit for the client connections. The accelerator operates in LAN environment here, with booster intercepting requests from the clients before sending them to the web server. The CPU utilization for the 10KB average object size is almost half of the base case, mainly due to the large decrease in the hardware and software interrupt processing overhead. The dependency on the object size is much more pronounced in this test, due to the larger weight of the connection open and close overhead compared to the base case.

Figure 8 shows the effect of avoiding connection open and close by using HTTP/1.1 persistent connections in addition to MTU increase shown on Figure 7. The scale of the Y-axis is further decreased 2.5 times on this graph. In this case, the booster multiplexes client connections over pool of persistent connections opened to the server. The effect of this optimization is most significant for the small object sizes. But even for the 10 KB objects the CPU utilization is again about half that of the previous graph. This graph also shows that the cost of hardware and software interrupts is now comparable with the other components.

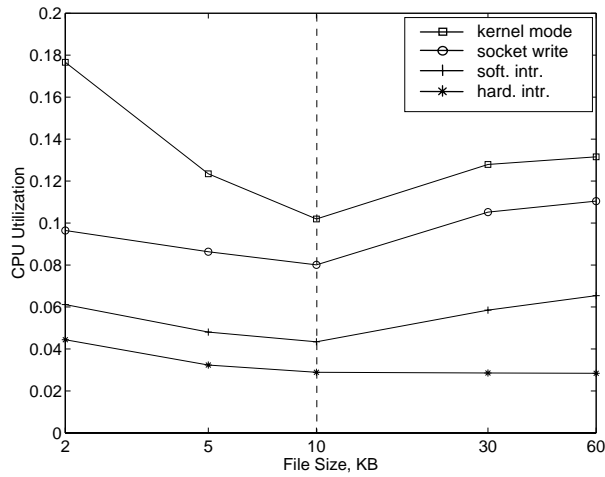


Figure 8. Cost breakdown vs. file size: persistent connections

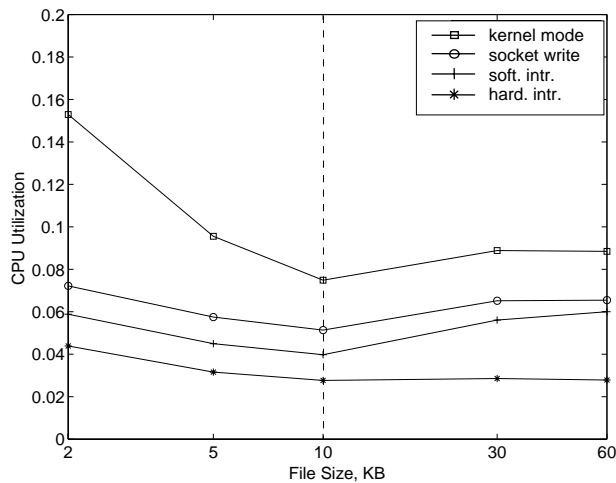


Figure 9. Cost breakdown vs. file size: no copying

Figure 9 shows the effect of avoiding copying, checksumming and packet management by enabling a packet cache in addition to the optimizations shown on previous two graphs. This optimization decreases the socket write component of the overall cost, which became the largest overhead after applying the two previous optimizations. The effect is most noticeable for large object sizes because of the amount of copied data. For 10KB objects, this optimization makes CPU utilization about two-thirds of the previous graph.

Finally, Figure 10 shows the effect of delaying acknowledgement from the booster. In this experiment, acknowledgements are sent by booster for every fourth data packet received. Again, the effect is most noticeable for large object sizes, where there is a larger fraction of acknowledgements available for delaying. This technique mostly decreases hardware interrupt cost. Its effect is again a reduction of two-thirds in the CPU utilization over that of the previous graph.

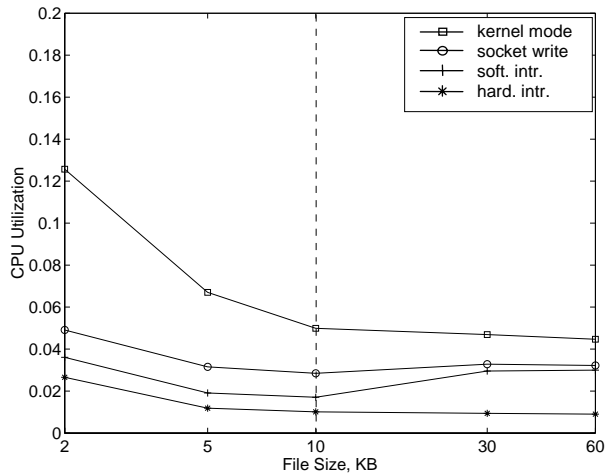


Figure 10. Cost breakdown vs. file size: delayed ack's

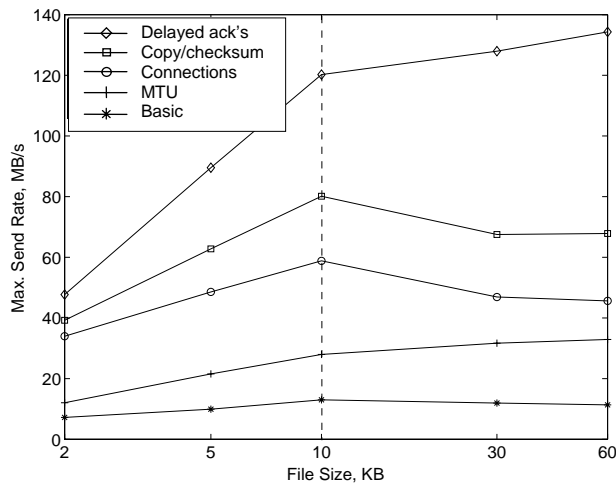


Figure 11. Maximum send data rate vs. file size

Figure 11 shows the extrapolated maximum achievable data rate assuming that CPU utilization scales linearly with load. Such assumption is based on the experiments with TUX described elsewhere [40], which show that utilization scales linearly with load until almost 95%. For a 10KB object size, the optimized configuration performs about 9 times better than non-optimized and can achieve about 120MB/s send data rate, or 12000 requests per second. For 60KB objects, the speed up is even higher: about 10 times.

To see what effect the optimizations have under a workload with the range of the object sizes, we measured accelerator performance under SURGE workload with 10KB average object size. Figure 12 shows the comparison of the cost breakdown under this workload and under the micro-benchmark workload with requests to a single 10KB object with a constant rate. The web accelerator uses all optimizations described above. The graph shows the same cost components as the previous graphs. The SURGE workload increases the processing overhead about 30% relative to the micro-benchmark

TABLE I. ACCELERATOR STATISTICS, SURGE WORKLOAD

	Accel. send rate, MB/s	# packets rcv. /s	# packets sent /s	# interrupts /s
Base	6.0	11967	12395	14788
Optimized	6.0	1102	4210	1381
Intr. coalescing	6.0	1105	4214	631

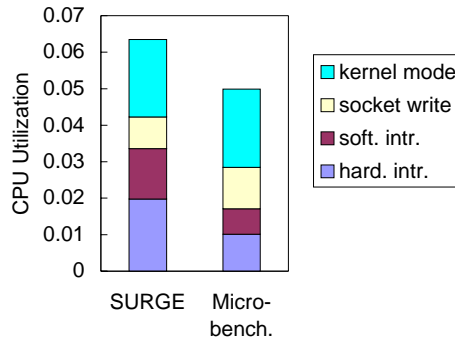


Figure 12. SURGE vs. micro-benchmark: all optimizations

workload, mainly due to the software and hardware interrupt processing. From Figure 5 (no optimizations) and Figure 12 (all optimizations) we can see that the performance improvement under the SURGE workload is about 9 times, approximately the same as for the micro-benchmark workload. The extrapolated achievable data rate is slightly lower: 96 MB/s or 9600 requests per second for 10KB average object size.

D. Interrupt Coalescing

Interrupt coalescing is used to decrease the cost of interrupt processing by processing several received packets during one interrupt. To evaluate the effect of interrupt coalescing for the web accelerator, the performance of the web accelerator was measured with coalescing enabled. Figure 13 gives the comparison of the cost breakdown with and without interrupt coalescing. The same SURGE workload was used as in the previous experiment. The graph shows that interrupt coalescing does not introduce noticeable savings in the web booster environment. This is because the primary target of the interrupt coalescing, decreasing the number of hardware interrupts, was achieved by different techniques in the web booster environment.

Table I presents experimental statistics for the SURGE tests with non-optimized, optimized and interrupt coalescing enabled configurations. The cost breakdown for these three experiments was shown on Figure 5 and Figure 13. From this table we can see that in the optimized configuration, the web accelerator receives almost 11 times fewer packets than in the base case. This decreases the number of interrupts processed also by almost 11 times, bringing it to 1381 interrupts per second for the given data rate. Interrupt coalescing halves this number, but it does not significantly decrease the processing overhead because the number of interrupts is already sufficiently low.

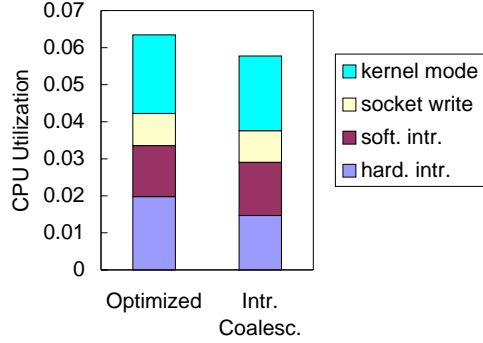


Figure 13. Effect of interrupt coalescing: all optimizations

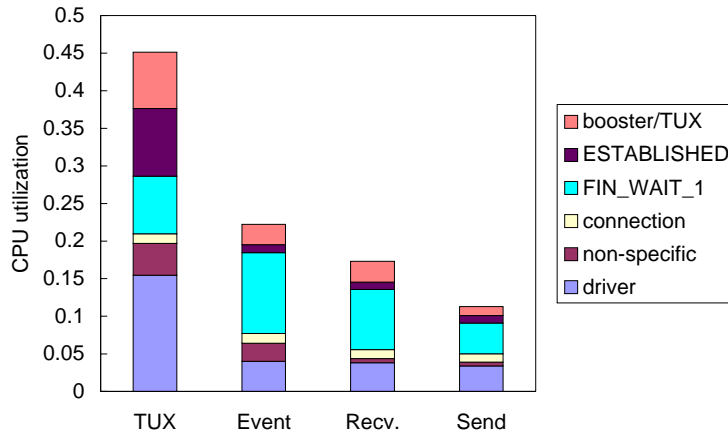


Figure 14. Booster cost breakdown comparison

E. Web Booster Performance

We measured the performance of the web booster under a simulated WAN environment. These experiments evaluate techniques for web booster performance improvements described in Section VI. In-depth description of the booster performance measurements can be found in Panteleenko and Freeh [41].

Figure 14 compares request processing cost breakdown for the TUX web server, and three booster configurations. The first one has only event polling enabled, the second one adds two receive processing optimizations and the third one adds two send processing optimizations. The SURGE tool generates the workload for the experiments with 10KB average object size. The MTU size is set at 536 bytes, the network delay at 200 ms and the connection bandwidth limit at 56 kbps. As previously, the send data rate is 6MB/s.

The cost of the booster processing in the base configuration (only event polling) is half that of TUX processing. The driver processing is decreased by almost a factor of three due to polling and streamlining passing packets to the protocol level. Another contribution is the difference in processing overhead between the booster module and the TUX kernel module. The four optimizations introduced in the Section VI together further decrease the processing cost by another factor of two to about one quarter that of TUX. Figure 14 shows that CPU utilization of the booster for 6MB/s send data rate 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.

TABLE II. REQUEST LATENCY

Accelerator end-user, ms	2220
Web booster + accelerator end-user, ms	2430
Accelerator with requests from booster, ms	10

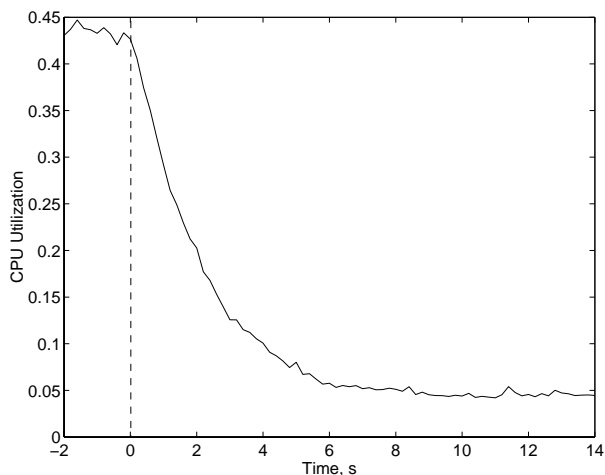


Figure 15. Acceleration activation transition function

F. End-user Latency

Introduction of the booster on the path between the clients and the server introduces additional latency. We measured request latency when the accelerator directly accepts client requests and when the booster is in the path. The latency is measured as the time from the client calling *connect* until the last byte of the response is received. Table II shows the end-user request latency for these two cases. Experiments used SURGE workload with 10KB average object size, 200 ms network delay, 56 kbps connection bandwidth limit and 536B MTU size. The table also shows the latency of the accelerator alone when it accepts requests from the booster. The latter is measured from the time the request was sent from the booster till the time the last byte of the response was received by the booster.

Table II shows that the request latency of the accelerator accepting requests from the booster is comparable with the timer interrupt tick, and about 200 times less than when it accepts requests directly from the clients. The main contributions to the end-user request latency are network delay and bandwidth limit. Introduction of the web booster in the path between the clients and the accelerator increases the latency by about 10%.

G. Activation Latency

The last set of experiments measures the latency of an activation of the acceleration techniques, which determines how fast the server can be offloaded. In these experiments the accelerator is switched from processing requests directly from the client machines to processing an optimized request stream from the booster. The booster is set as a router and initially all client requests are routed through the booster to be directly processed by the accelerator. At the specified time, client machines change the destination IP address of requests to the booster machine and booster starts processing these requests sending them to

the accelerator with all optimizations enabled. All requests that were in progress at the moment of the switch are processed as previously with the booster serving only as a router.

Figure 15 shows the transition function for CPU utilization. Unlike the rest of the experiments in this paper, the accelerator CPU utilization is measured over 200 ms time intervals. The CPU utilization presented on the graph is the average of 10 experiments. The network parameters for the client to the booster link are set to the base values of 536B MTU size, 200 ms network delay and 56 kbps bandwidth limit, which directly affect accelerator processing in the initial stage of the experiments when the booster serves only as a router. The booster is activated at the time 0 on the graph. Most of the processing during the few seconds after the activation handles un-optimized client requests that have not been finished before the booster is activated. The graph shows that the CPU utilization reaches about 40% of the original value at 2 sec after the activation.

VIII. CONCLUSION

This paper described a novel approach to offloading of a web server: changing network traffic between clients and a server. The traffic is modified by the booster appliance located in front of a web site. This approach decreases overhead of request processing up to nine times for both static and dynamic resources.

Six acceleration techniques are based on a study of web server performance. These techniques do not migrate or distribute any state from the web server and do not modify the server software. These techniques are: (i) increasing MTU size of the connections to a web server, (ii) avoiding connection open and close, (iii) delaying client acknowledgements, (iv) performing request processing in the kernel, (v) avoiding data copying and checksumming, and (vi) avoiding packet creation overhead. These techniques decrease the hardware interrupt overhead to the point where the interrupt coalescing does not introduce significant improvement.

The paper described a web accelerator software module running on the web server that enables last three of the above optimizations. It caches the requested documents in kernel in form of TCP packets avoiding the cost of copying, checksumming and packet creation for subsequent requests. The paper also described the implementation of the web booster appliance and four optimization techniques of the network protocol processing of the booster. These techniques allow the booster to forward requests at a rate four times greater than the web server process them on the same hardware platform

The results in this paper indicate that the effectiveness of the acceleration techniques depends on the properties of the client connections. If the client population for public web servers changes from predominantly dial-up users to users with high-speed Internet connections with large MTU sizes and the clients start to use HTTP/1.1 protocol so that the connection open and close overhead is eliminated, the base case when the accelerator directly processes the client requests will have lower CPU utilization and consequently there will be smaller potential speed-up by applying the optimizations.

There are several possible extensions to the proposed architecture. Web booster functionality can be easily extended with layer-7 request distribution and load balancing. Most of the required expensive processing necessary for layer-7 aware request forwarding, such as terminating client connections and obtaining request URL, is already done at the booster. The logic required for this additional functionality has low overhead compared to the rest of the processing [9].

Another possible extension for the booster architecture can provide a fine-grained request admission policy. Most of the existing commercial and research web servers have coarse resource management. For example, because the significant part of the network processing is done in interrupt context, the cost associated with it is attributed to the currently interrupted process. This leads to denial-of-service attacks when a malicious principal overloads the system with processing. The current design of general-purpose operating systems makes this problem hard to solve [11]. The web booster may provide a way of enforcing the admission policy by filtering or routing requests that are sent to the web server based on the client identity, enabling more fine-grained resource accounting. This would also enable client

differentiation based on business criteria such as client subscription level. Because the web booster itself can be built using specialized software and/or hardware, the problem of fine-grained resource allocation is easier to solve there than in the general-purpose operating system.

In a high-density web hosting environment, heat dissipation by thin web server “blades” poses a problem. Offloading most of the server computation to a specialized booster appliance can alleviate this problem. Such offloading decreases necessary CPU capacity of the servers enabling less powerful, energy-efficient architectures and/or it decreases the current heat dissipation by lowering the CPU utilization.

REFERENCES

- [1] Akamai Technologies, Inc. <http://www.akamai.com>.
- [2] Alacritech Inc. <http://www.alacritech.com>.
- [3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. Technical Report 96-011, Boston University, June 1996.
- [4] Alteon Web Systems Inc. ACEdirector. <http://www.nortel.com>.
- [5] Alteon Web Systems Inc. Alteon SSL accelerator. <http://www.nortel.com>.
- [6] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandervoort, 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)*, Oct. 1997.
- [7] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
- [8] 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.
- [9] 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*, June 2000.
- [10] 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.
- [11] G. Banga, P. Druschel, and J. C. Mogul. Better Operating System Features for Faster Network Servers. In *Proceedings of the Workshop on Internet Server Performance (held in conjunction with ACM SIGMETRICS '98)*, June 1998.
- [12] G. Banga, J. C. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of 1999 USENIX Annual Technical Conference*, 1999.
- [13] P. Barford and M. E. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of Performance '98/ACM SIGMETRICS '98*, 1998.
- [14] BoostWorks, inc. <http://www.boostworks.com>.
- [15] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM '99*, March 1999.
- [16] R. Caceres, F. Dougliis, A. Feldmann, G. Glass, and M. Rabinovich. Web Proxy Caching: The Devil is in the Details. In *Proceedings of Workshop on Internet Server Performance*, June 1998.
- [17] M. Carson. NIST Net. <http://www.antd.nist.gov/nistnet>.
- [18] A. Chandra and D. Mosberger. Scalability for Linux Event-Dispatching Mechanisms. In *Proceedings of 2001 USENIX Annual Technical Conference*, June 2001.
- [19] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K.J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, Jan. 1996.
- [20] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *Proceedings of 2001 USENIX Annual Technical Conference*, June 2001.
- [21] Cisco Local Director. Technical White Paper, Cisco Systems Inc., 1998.
- [22] Digital Island, Inc. <http://www.digitalisland.com>
- [23] F. Dougliis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of Change and Other Metrics: a Live Study of the World Wide Web. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.
- [24] Giganet, Inc. Giganet cLAN Product Family. <http://www.giganet.com/products>.
- [25] 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.
- [26] IBM Corporation. IBM Interactive Network Dispatcher. <http://www.ibm.com/software/network/dispatcher>.

- [27] Infiniband Architecture Specification. Vol. 1&2. Rel. 1.0. http://www.infinibandta.org/download_spec10.html.
- [28] P. Joubert, R. B. King, R. Neves, M. Russinovich, J. M. Tracey. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *Proceedings of 2001 USENIX Annual Technical Conference*, June 2001.
- [29] D. Krager, T. Leighton, D. Lewin, and A. Sherman. Web Caching with Consistent Hashing. In *Proceedings of the 8th Int. World Wide Web Conference*, May 1999.
- [30] 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*, June 1998.
- [31] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the ACM SIGMETRICS '97 Conference*, June 1997.
- [32] Microsoft Corporation. Installation and Performance Tuning of Microsoft Scalable Web Cache (SWC 2.0). <http://www.microsoft.com/technet/iis/swc2.asp>.
- [33] 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.
- [34] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Transactions on Computer Systems*, 15(3), Aug. 1997.
- [35] 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.
- [36] NetScaler. <http://www.netscaler.com>.
- [37] 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*, Oct. 1998.
- [38] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, Feb. 1999.
- [39] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.
- [40] 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.
- [41] V. V. Panteleenko and V. W. Freeh. Optimizing TCP Forwarding. Technical Report TR-02-03, University of Notre Dame, July 2002.
- [42] Redline Networks. <http://www.redlinenetworks.com>.
- [43] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. In *Computer Communication Revue*, 27(2), Feb. 1997.
- [44] 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*, March 1998.
- [45] H. V. Shah, D. B. Minturn, A. Foong, G. L. McAlpine, R. S. Madukkarumukumana, and G. J. Regnier. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01)*, March 2001.
- [46] The Standard Performance Evaluation Corporation. SPECWeb96/SPECWeb99. <http://www.spec.org/osg/>.
- [47] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of the 19th IEEE Int. Conference on Distributed Computing Systems*, May 1999.
- [48] TUX Web Server 2.0. <http://www.redhat.com/products/software/tux>.
- [49] A. van de Ven. kHTTPd Linux HTTP accelerator. <http://www.fenrus.demon.nl>.
- [50] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, and H. M. Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP '99)*, Dec. 1999.