Instantaneous Offloading of Transient Web Server Load[1]

Vsevolod V. Panteleenko and Vincent W. Freeh
Department of Computer Science and Engineering
University of Notre Dame
{vvp, vin}@cse.nd.edu

Abstract

A modern web-hosting site is designed to handle load that is sometimes an order of magnitude greater than the average load. Such a site can be expensive and is underutilized most of the time. We describe a design and performance study of the *web booster* architecture, which reduces web server load during peak periods. A web booster, inserted between client and server, instantaneously decreases server processing costs for requests to static documents while keeping the processing on the origin server. Fast reaction to the load change and the fact that the booster does not hold resources during inactive periods allows a web booster to be time-shared among multiple web domains that physically reside on different servers.

A *web accelerator* module on a web server interacts with the web booster. It uses network packet caching, optimizing the TCP protocol and offloading of computation to increase the rate at which a server can process client requests. This paper describes the web booster architecture and our implementation of the accelerator, which decreases the web server load by more than a factor of three.

Keywords: web server, load balancing, web booster.

1 Introduction

Traffic to a web server domain is unpredictable and very peaky. Request rate peaks can be up to ten times the average rate [Ste96]. To avoid losing or alienating clients, web-hosting sites are overbuilt to handle the peak load. While some of the sites are built as web server clusters that host multiple web domains, many of the sites require each domain to reside on dedicated servers for security or other reasons.

This paper introduces the *web booster* architecture, a new load distribution mechanism for web hosting sites that utilize dedicated servers for each web domain. This mechanism temporarily and instantaneously decreases the request processing cost for static and cached dynamic documents on a web server in response to transient peaks in load (Figure 1). During normal operation, client requests are delivered directly to the servers of the target domain. When the load on a domain approaches the

---

[1] Version of this technical report was published in *Proceedings of Sixth International Workshop on Web Caching and Content Distribution (WCW '01),* June 2001

maximum capacity of the servers, client requests are redirected to a web booster, which pre-processes requests and forwards them to the corresponding servers in an optimized request stream. On a web server, the *accelerator* kernel module receives requests from the booster. It can service requests itself at a decreased processing cost or pop the request up to the user-level HTTP server for general processing. Replies from the web server are sent back to the booster, which delivers them to the clients.

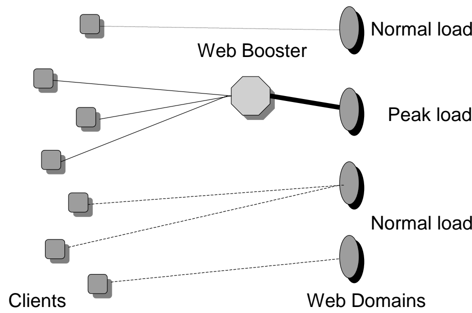


Figure 1. Web booster

The fact that the load may be decreased instantaneously enables effective time-sharing of the web booster among multiple web domains that have low correlation among load peaks. A web booster serves one or few overloaded domains at a time, which prevents it from become a bottleneck for the hosting site. The actual document processing is kept on the origin web servers.

Compared to accepting and processing requests directly from clients, using a web booster reduces the load on the origin server in several ways. First, the web booster communicates with a web server using an optimized TCP/IP, which decreases the network protocol processing cost on a web server. Second, the accelerator module caches documents in the form of network packets avoiding data copying, checksumming and network buffer management. Third, some request processing may be migrated to the web booster, such as parsing HTTP headers.

Caching is a good mechanism for permanently expanding the capacity of a web server. But it is not a good solution for providing temporary capacity needed for handling transient loads because this mechanism have a long "warm-up" period while acquiring the resources necessary to share the load. Additionally, a cache has to be kept consistent with the original domain, whereas in the booster model, the original domain retains control of all document content. Furthermore, because all requests go to the origin server, it has the final decision as to how to handle the request.

We implemented the web accelerator as a kernel module that runs on top of the Linux operating system without any modification to the operating system or device drivers. We also used a modified SURGE [BC98] web workload generator to emulate the web booster. Experiments show that the web booster architecture decreases web server load by more than factor of three.

Next section overviews related work. Section 3 gives the performance measurements for the user-level web server and identifies techniques we used to decrease the request processing cost on a web server. Section 4 describes the design of the accelerator software which resides on the origin sever. We measure the performance effects of using the accelerator in Section 5. Section 6 discusses

the design issues for the web booster. The following Section 7 talks about possible extensions and the last section concludes.

2 Related Work

Several existing systems use the Internet infrastructure to migrate processing from subscribed web servers. One such commercial system, Akamai, provides a set of high performance web servers distributed throughout the Internet [Aka]. Subscribed web servers upload large files, such as multimedia, on these dedicated servers, which in turn deliver them to the clients. Origin web servers accept client requests for some documents, while Akamai servers perform file transfers that require high processing and network resources. In this way, web providers completely control their web sites without owning or maintaining high-performance hardware.

Another popular way to offload web servers is using proxy caching and web accelerators (different from our accelerator). The former are used on behalf of clients and might cache documents requested from any server. The latter are used to cache documents only for a specific set of servers. There is a large body of work that studies different aspects of web proxy caching such as web traffic characteristics from the viewpoint of the proxy caches [ABC+96] [BCF+99] [CDF+98] [DFK+97] and cooperative caching [WVS+99] [CDN+96] [KLL+99] [TDV+99]. These studies show that the effectiveness of caching is limited by such factors as document changes and number of non-cacheable documents.

There has been a lot of research done in the area of load balancing and request routing inside a single web server cluster. Several works analyze content-based request distribution policies [PAB+98] [ZBC+99]. Others study mechanisms for redirecting requests to a particular web server, including DNS-based [Bri95], TCP splicing [DKM96] [CRS99] [YL99], TCP hand-off [ADZ99] [PAB+98] and distributed redirection [ASD+00].

There are multiple studies that analyze the performance of web servers and the effects of different design choices. Pai, et al., looked into performance of different server architectures in context of one implementation: the Flash web server [PDZ99]. Lava caching server shows the effect of using a specialized operating system and building the server as integrated software [LPJ+98]. Two in-kernel web server daemons [Khttpd] [TUX] attempt to decrease the overhead of user – kernel context switches. The latter also implements zero-copy send operation by using gather/scatter network buffer implementation and hardware checksumming. Levi, et al., describes the design and performance of a web accelerator deployed as the front end to a set of web servers [LIS+99].

Several works analyze the cost of different operations of a busy web server. Some of these studies use a simulated workload [HMS98] [BC98], while others study server behavior under real-life workload [MRG97] [CDN+96] [BM98]. These studies show the importance of reproducing such workload characteristics as request inter-arrival time, network delays, and a large number of simultaneously open connections.

There are several areas in operating systems design that are important for web server performance. U-Net proposed a user-level interface that virtualizes the network datalink layer and allows the implementation of network protocols at the user level [EBB+95]. Its design makes it possible to tailor the protocols and network buffer management for a particular application. Active Messages integrate communication with computation by allowing a received message to run user

code as the handler on the stack of the running computation [ECG+92]. This avoids scheduling overhead for such computation and unnecessary data copying. IO-Lite introduces a unified I/O buffering and caching system [PDZ99a]. It allows applications, inter-process communication, the file system, the file cache and the network subsystem to share a single physical copy of data. Protection and security are maintained through a combination of access control and read-only sharing.

3 HTTP Request Processing

Before describing the web accelerator architecture, we present a brief overview of how request processing is performed by a regular HTTP server running on top of a general-purpose operating system. Quantitative analysis of the request processing cost, which is based on our experiments, will help us to identify possible targets for decreasing such cost.

The cost of request processing on a general-purpose web server may be roughly subdivided into the four components described below.

User-level processing. The processing cost includes HTTP header parsing, header generation, request decoding and bookkeeping, and data cache management (if a user-level cache is used). It is accomplished in the context of one or more HTTP server processes or threads.

Kernel-level, process context. Most data sending is accomplished in this context. This includes copying data from user level into network buffers, calculating checksums, and sending or queuing data to the network device driver. The copy of the received data from the buffers to the user level is accomplished in this context as well. Another component of request processing performed in this context is the small part of the connection management, mostly related to *accept* and *close* system calls. The event synchronization necessary for an event-based web server, and usually performed by *select* or *poll* system calls, is done here too.

Hardware interrupts. A network device driver processes received network packets in the hardware interrupt context. It also uses hardware interrupts to release the packets that have been sent by the network adapter.

Software interrupts (deferred procedure calls)**.** The remainder of the network protocol processing is accomplished in software interrupts. This work includes most of the receive processing, checksumming received packet, sending acknowledgements to the received data, retransmitting, and sending queued packets. It also includes most of the connection management processing.

To identify possible ways to decrease the request processing cost, we study the performance of the Flash web server, one of the fastest servers reported in the literature that runs on top of many general-purpose operating systems [PDZ99]. Most of the objects in our measurements are delivered from the main memory.

The test platform is a 650MHz Intel processor with 512MB of the main memory running Linux v2.4 operating system. The Flash web server was driven by the workload generated by SURGE toolkit [BC98]. We varied the number of requests per second to obtain data that covers the range of the server load. The measured data were approximated by a linear polynomial using the least-square method. We will give more detailed description of the experimental setup, workload characteristics

and the measurement methodology in Section 5, which describes the performance measurements of the accelerator using the same techniques.

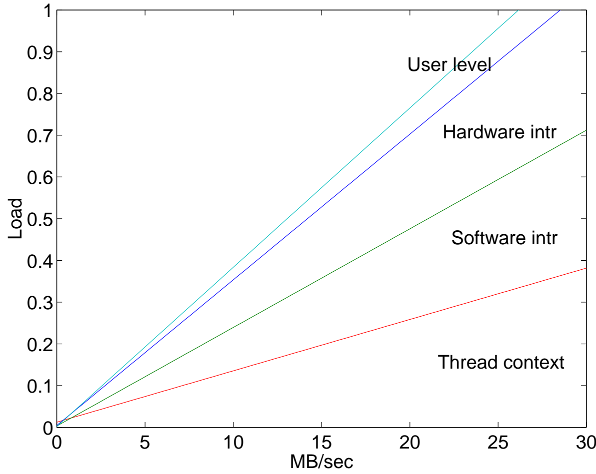


Figure 2. Cost breakdown: Flash HTTP server

Figure 2 gives the graph of approximated data for different cost components. From this graph, we can see that the major cost of request processing is related to network protocol overhead. About one-third of the overall cost is due to the hardware interrupts. This cost does not depend on the size of the packets processed, but rather on the number packets, because the device driver processing does not touch the packet data. Additionally, more than one-third of the overall cost is spent in the kernel mode in process context, which mostly involves data copying, checksumming and buffer management.

To decrease the overall cost of the request processing, we target the following cost components.

- **Send data copying and checksumming.** Using network packet caching in the accelerator makes it possible to perform these operations only once on a cache miss.
- **Network buffer management.** The packet cache also avoids network buffer management overhead.
- **Connection management.** Opening several network connections from the booster to the origin server and sending all requests through them eliminates connection management overhead and decreases the corresponding cost of hardware and software interrupts.
- **Number of network packets processed.** Delaying acknowledgements from the booster further decreases the number of network packets processed and the corresponding cost of hardware and software interrupts.

The next section describes the design of the accelerator that enables these optimizations.

4 Web Accelerator

The web accelerator resides on the origin web server and optimizes HTTP request processing. It is a kernel module that is able to completely process HTTP requests for static documents. Requests to dynamic documents or to those that require special processing are redirected to a regular HTTP server that runs in user space. The web booster communicates with the origin server using TCP/IP. The accelerator is active even when web booster is not used, and the origin server accepts requests directly from clients.

The accelerator uses three techniques to decrease the cost of packet processing. First, it avoids data copying, checksumming and network buffer management by caching documents as the network packets. Second, it communicates with the web booster using optimized TCP, which avoids network connection/teardown management and decreases the number of processed network packets. Lastly, it may offload some of the HTTP request processing to the web booster.

Figure 3 shows the design of the accelerator. It consists of the 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. 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 user-level the HTTP server to read requests.

The MTU of clients can vary greatly. For example, according to the study of the web server under real workload [Ste96], only 32% of all clients advertised MSS value of 1460, which corresponds to Ethernet MTU, and 60% of them advertised a value less than 1024. The packet caching technique only can be used within a range of the MTU's that are used for connection to the web booster or clients. If the size of the packets for a requested document is larger than MTU of the connection, the packets have to be fragmented, which implies copying and checksumming. If the size is smaller, transmission of such packets unnecessarily increases the number of processed network packets and the corresponding cost of request processing. When the accelerator communicates with a web booster, the packet cache can be built for the single MTU of the booster connections. If the accelerator receives requests directly from clients, it deals with a wide range of MTU, which makes the packet cache significantly less efficient.

The web accelerator uses an unmodified TCP/IP stack to communicate with the web booster. The booster opens several TCP connections and sends all requests through them. Such set up avoids connection open and close overhead, which is usually incurred when clients send requests directly to the accelerator. Also smaller number of concurrent connections decreases the cost of packet de-multiplexing, which may significantly affect the overall performance [BM98].

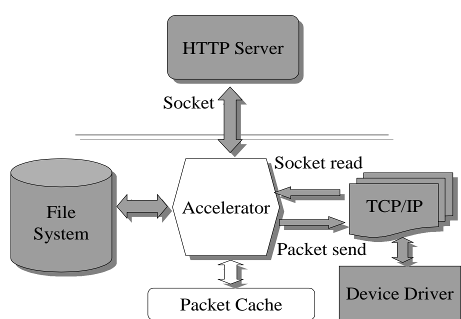


Figure 3. Accelerator design

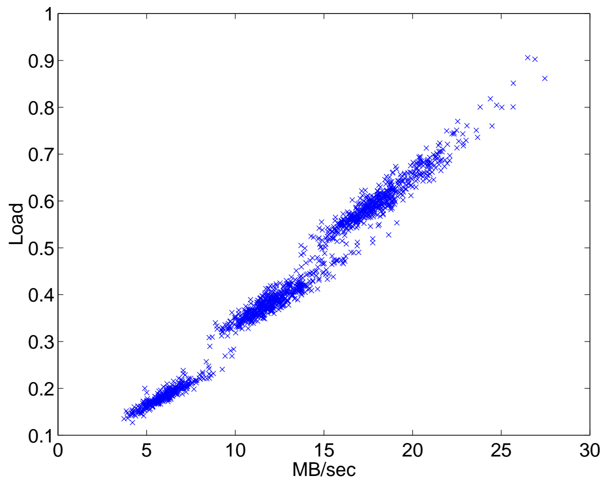
The web booster delays acknowledgements in order to decrease the overhead of network packet processing. When a booster and an origin server are connected in a local area network, the rate of packet loss is low. In this case it is possible to send acknowledgements from the booster for the sent data less frequently than for every two packets, as the TCP specification requires. Such technique significantly decreases the number of packets processed on the origin server. This in turn decreases the number of hardware and software interrupts and the corresponding cost of the request processing. The TCP module of the web booster has to be modified to utilize this method, but not the origin server.

One potential problem with this approach is the slow start of the origin server's TCP. Delaying the acknowledgments during this phase would prevent the origin's server TCP to leave slow start quickly and, by this, it would limit the throughput of the connection. Therefore, the web booster enables delayed acknowledgements only after the slow start.

Although the accelerator is able to completely process HTTP requests, it may offload some computation for HTTP header parsing to the web booster. In this case, the web booster parses and extracts the requested URL from the HTTP request, but it sends this URL together with the original header. The accelerator then decides whether to process the request or to redirect it to the user-level HTTP server, in which case the original HTTP header is used. Previous section showed that the cost of the HTTP request parsing is not significant compared to the overall processing cost, so the usefulness of this technique depends on whether HTTP header parsing on the web booster is required for other reasons, such as those described in Section 7.

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

- **Memory management.** The accelerator requires non-paged memory allocation that is possible to use as network buffers.
- **Packet queuing.** The TCP implementation must export an entry point that allows one to send or queue built network packets, avoiding the regular data copying and packet creation mechanism.



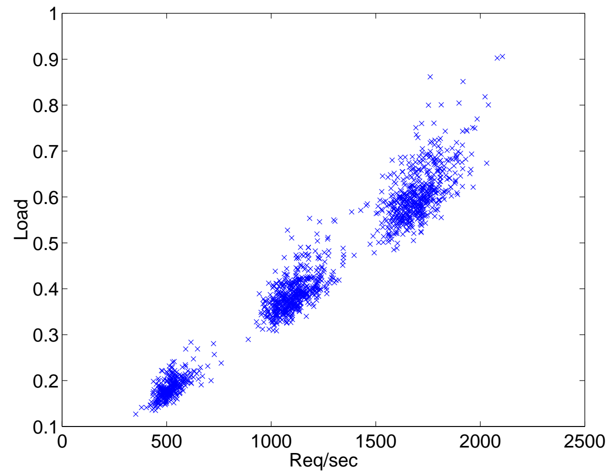


Figure 4. Load vs. send data rate

Figure 5. Load vs. request rate

- **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, which was used as our implementation platform.

In the current design, cached documents are used only by the accelerator and are cached directly as the network packets. When files are read from the file system, the page cache is avoided, so that there is only one cached copy of the file in the memory. Nevertheless, we envision that there might be a situation when the file data have to be both available to the accelerator in forms of network packets and mapped to a user-level process as memory pages. Using two caches is possible but doubles the memory consumption.

It is possible to have only one copy of the file data in the memory if the operating system and the network device driver support multiple non-contiguous buffers for one packet (gather/scatter). In this case a page cache where the file is read is used to create a network packet consisting of two non-contiguous buffers: the packet header built in a separate memory pool and the packet data that are just part of the unmodified page. This page may be also directly mapped into user-level space as read-only page if necessary.

Although some of the processing optimizations described in this section are applicable to the stand-alone configuration when the origin web server accepts requests directly from the clients, the web booster is required in order to:

- avoid connection management overhead,
- use delayed acknowledgement technique,
- fully utilize the packet cache (because a single MTU is used), and
- offload some request processing to the web booster.

5 Performance Results

We used Linux kernel v. 2.4 as our implementation and test platform. The accelerator was implemented as a kernel module. For experiments, we used a machine with Intel 650MHz processor equipped with 512MB of main memory and connected to the test clients with three 100Mbps Ethernet cards.

The SURGE simulation toolkit [BC98] generates the workload. It uses a statistical model to simulate the inter-arrival time, distribution of document sizes, request sizes and temporal locality of the real life HTTP clients. The number of so called "user equivalents" defines the level of the workload, which are modeled after the real life clients that generate series of HTTP requests separated by off-time.

To measure the cost of request processing in different parts of the kernel and the accelerator, we wrote a kernel profiling tool that uses Intel processor hardware counters to precisely measure the time spent in different parts of the kernel. It takes into account process context switches and interrupts when measuring the time.

5.1 Measurements

We ran four sets of tests in which we enabled different optimization techniques described in previous section. Each set consisted of three runs with different number of SURGE "user equivalents": 1200, 2400 and 3600. Because the SURGE toolkit generated workload that has some distribution over time, we measured the load and other test parameters averaged over 0.5-second intervals.

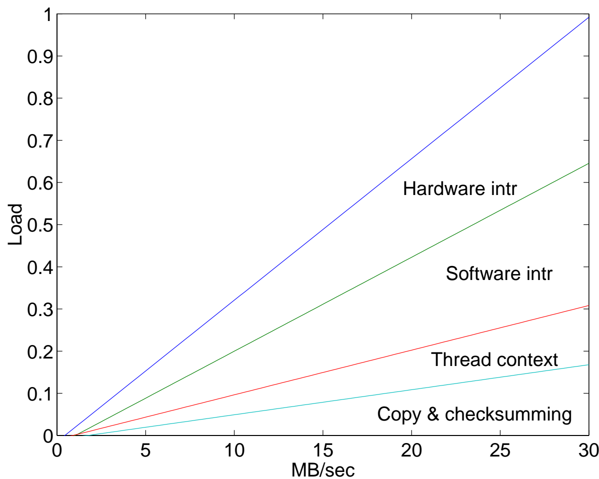


Figure 6. Cost breakdown: unoptimized

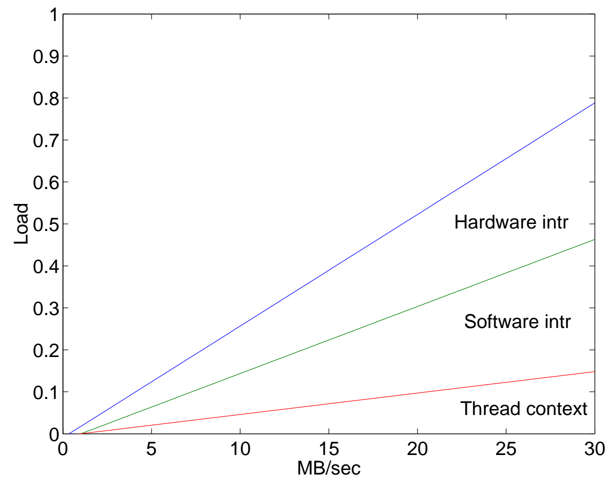


Figure 7. Cost breakdown: packet cache

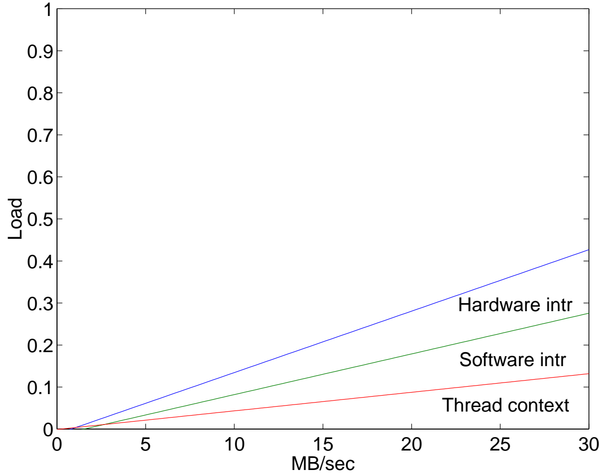


Figure 8. Cost breakdown: no connections

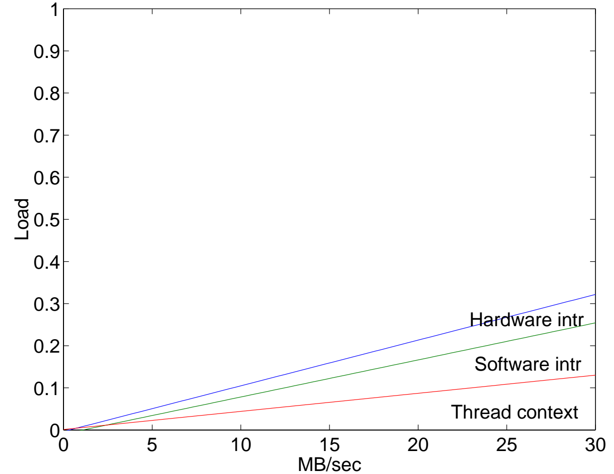


Figure 9. Cost breakdown: delayed ack's

	Unoptimized	*Optimized*
Average request size	11.4 KB	11.4 KB
Sent packet / request	12.9	9.31
Received packet / request	9.41	1.41
Hardware intr / packet	0.51	0.22
Software / hardware intr	0.53	0.54

Table 1. Test parameters

Table 1 shows different test parameters for the configuration with all and none of the optimization enabled. The optimized configuration experiences about one-fourth as many hardware interrupts per requests as the unoptimized configuration. This reduction is primarily because there are almost seven times fewer received packets processed in optimized configuration.

Figure 4 shows the measurements of the load plotted against the send data rate for the accelerator configuration with no optimizations enabled (which is comparable to a kernel HTTP server). In this case, a new TCP connection is opened for every request. Each point corresponds to 0.5-second interval. Figure 5 shows the same data plotted against request rate. We can see that the data in the first figure grouped closer to the linear graph, while in the second graph data are more scattered. This indicates that the send operation is the major contributing factor for the overall processing cost, while the additional overhead per request does not have significant influence. Furthermore, when the network cards are nearly saturated (30MB/sec), the load on the server approaches 100%.

We measured the contributions of hardware and software interrupts to the overall processing cost. Further we break down the request processing cost in the context of the accelerator thread into the cost of copying and checksumming and the cost of the rest of the processing. The latter includes

a small part of the cost of connection management, which is mostly accomplished in context of software and hardware interrupts.

Figures 6, 7, 8 and 9 show the approximation of the measurement data with a linear polynomial using least-square method. Figure 6 is the same configuration with no optimizations as discussed above. Hardware interrupts, software interrupts, and copying-checksumming contribute the most to the cost of the processing. Figure 7 shows the configuration when the packet cache is enabled. This optimization eliminates copying and checksumming and decreases the thread context processing cost. Its overall processing cost is approximately 75% of the unoptimized configuration.

The next test enables connection management optimization in addition to packet caching, which is shown in Figure 8. The connection cost performed in the context of an accelerator thread is a minor amount of load. However, this greatly reduces interrupts, which cuts the cost of hardware and software interrupts in half.

Finally, Figure 9 shows the effect of delaying acknowledgements sent from the booster to the server. This shows the effect of all the optimizations discussed in this paper. The use of delayed acknowledgements again halves the cost of hardware interrupts.

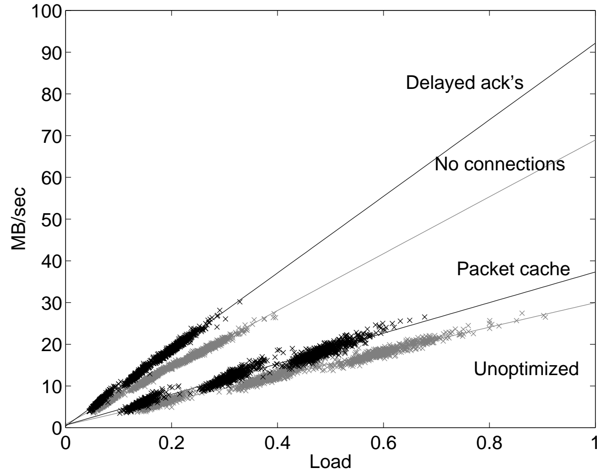


Figure 10. Maximum data rate

Figure 10 shows the overall load measurements for all four configurations discussed above. This graph plots the data rate vs. load and uses linear least-square approximation that is superimposed on the measured data. The axises of the graph are reversed from the previous graphs, so in this graph a steeper curve is better (more optimizations enabled). The graph shows that on the current test platform the extrapolated maximum data rate is about 90 MB/sec, which is about 9000 req/sec for 10KB objects. This is about 3 times higher than the rate for unoptimized configuration. Recall that unoptimized configuration performs slightly better than Flash user-level web server.

We compared the performance of the accelerator to the performance of the TUX web server [TUX] built as an in-kernel module. This server also avoids data copying and checksumming during data send operation but relies on gather/scatter packet buffer support, provided by Linux 2.4 kernel and some network device drivers, and on hardware packet checksumming. We run tests with TUX enabling the same types of optimizations as with the accelerator including delivering requests through a small number of permanently opened connections and delaying acknowledgements. TUX performance is similar to that of the accelerator, with less than 5% difference. The accelerator design

does not require any special support from the operating system or network card such as hardware checksumming. In fact, it was originally implemented on 2.2 Linux kernel without any such support.

6 Web Booster

The web booster has to terminate client TCP connections and route data from/to clients on separate TCP connections opened to web servers. This requires handling double the traffic of a served domain, because the booster sends to the clients approximately the same amount of data as it receives from the origin server. At the same time, the web booster does not have to scale with the size of the hosting site, since it is used only for one or few overloaded domains at a time. In fact, the web booster can be a more powerful and expensive machine than any of the domain servers since the whole site shares its cost. The booster can be easily replicated to increase the maximum throughput.

Unlike the web accelerator, which has to run within a general-purpose operating system, the web booster is a dedicated machine. This fact allows the customization of the operating system, network device drivers and adapters. The web booster has to contain the following functional components.

- Run-time environment exporting basic memory management, scheduling and interrupt processing functionality.
- Network buffer management facility that enables in-place packet processing where possible.
- TCP/IP stack that is integrated with the buffer management facility.
- Network device driver integrated with the protocol stack.
- HTTP parsing engine.

To handle the network traffic requirements of the web server, the web booster software can use a couple of techniques. First, it can use in-place packet processing. Many modern network adapters enable non-contiguous (gather/scatter) packet buffers and hardware checksumming. The device driver should utilize these features to build new client packets out of the packets received from the web server with possibly different MTU without doing copying and software checksumming. Second, the web booster can use network adapter polling. Polling is more efficient than processing interrupts when a system is busy.

Pai et al. [PAB+98] describe layer-7 switch that implements similar functionality to what web booster has to implement. They indicate that such a switch implemented on general-purpose hardware and software can handle the traffic of approximately ten web servers on typical web workload.

7 Possible Extensions

There are several ways of extending the proposed model. First, a web booster can handle secure channels between the web servers and the clients. If secure channel is required between the booster and the origin server, the packet cache stores already encrypted data. It would be decrypted at the booster and encrypted for each client. Caching encrypted data is not useful for direct client-server transfers because each client has a different key.

Second, most of the existing commercial and research web servers have coarse resource management. 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 [BDM99]. The web booster may provide a way of enforcing the resource allocation policy by filtering requests that are sent to the web servers, enabling more fine-grained resource accounting. Because the web booster itself can be built using a customized operating system, the same problem could be easier to solve there than in the general-purpose operating system.

8 Conclusion

In this work, we described the web booster infrastructure that is used as a shared resource by web domains residing on separate servers to temporarily offload the request processing from the web domain that experiences the peak load. This makes it possible to plan for a smaller infrastructure that handles the same load as a more expensive and often under-loaded infrastructure. To achieve the above goal we: (i) introduced the web booster model that decreases the HTTP request processing costs on the origin web server during peak load without removing processing from it; (ii) measured the performance of one the fastest known web server that runs on top of a general-purpose operating system and identified that the main cost of the request processing is related to the data copying / checksumming and interrupt processing; and (iii) designed, implemented, and measured the performance of the accelerator software, which decreases server load by more than factor of three when compared to direct client request processing.

Acknowledgements

We would like to thank David Cohn for his contributions to the idea and comments on a draft of this paper. The IBM University Partnership Program Award and the JPL HTMT project sponsored this work.

References

[ABC+96] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. Technical Report 96-011, Boston University, June 1996.

[ADZ99] 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.

[Aka] Akamai Technologies, Inc. http://www.akamai.com.

[ASD+00] 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.

[BC98] 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.

[BCL+98] A. Bestravos, M. Crovella, J. Liu, and D. Martin. Distributed Packet Rewriting and its Application to Scalable Server Architectures. In *Proceedings of the 6th International Conference on Network Protocols*, Austin, TX, Oct. 1998.

[BCF+99] 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.

[BDM99] 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 (WISP)*. Madison, WI, June 1998.

[BM98] G. Banga and J. C. Mogul. Scalable Kernel Performance for Internet Servers Under Realistic Loads. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[Bri95] T. Brisco. DNS support for Load Balancing. Technical Report RFC 1974, Rutgers University, April 1995.

[CDF+98] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web Proxy Caching: The Devil is in the Details. In *Workshop on Internet Server Performance*, June 1998.

[CDN+96] 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*, San Diego, CA, Jan. 1996.

[CRS99] 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.

[DFK+97] F. Douglis, 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.

[DKM96] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, Feb. 1996.

[EBB+95] T. von Eicken, A. Basu, V. Buch, W. Vogels. U-Net: A User Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, Dec 1995.

[ECG+92] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[HMS98] J. C. Hu, S. Mungee, and D. C. Schmidt. Techniques for Developing and Measuring High-Performance Web Servers over ATM Network. In *Proceedings of the INFOCOM '98 Conference*, San Francisco, March - April 1998.

[Khttpd] Khttpd Linux HTTP Accelerator. http://www.fenrus.demon.nl.

[KLL+99] 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.

[LIS+99] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias. Design and Performance of a Web Server Accelerator. In *Proceedings of IEEE INFOCOM'99 Conference*, New York, NY, March 1999.

[LPJ+98] J. Liedtke, 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.

[MRG97] 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.

[PAB+98] 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.

[PDZ99] 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.

[PDZ99a] 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*, New Orleans, LA, Feb. 1999.

[Ste96] W. R. Stevens. TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NTTP, and the UNIX Domain Protocols. Addison-Wesley, 1996.

[TDV+99] 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.

[TUX] TUX Web Server 2.0. http://www.redhat.com/products/software/tux.

[WVS+99] 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),* Kiawah Island, SC, Dec. 1999.

[YL99] 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.

[ZBC+99] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. HACC: An Architecture for Cluster-Based Web Servers. In *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.