Improving Web Server Performance by Network Aware Data Buffering and Caching

Sourav Sen[1] * and Y. Narahari[2]

[1] Hewlett Packard India Software Operations
29, Cunningham Road, Bangalore - 560 052, India
souravs@india.hp.com
[2] Department of Computer Science and Automation, Indian Institute of Science
Bangalore - 560 012, India
hari@csa.iisc.ernet.in

Abstract. In this paper we propose a new method of data handling for web servers. We call this method *Network Aware Buffering and Caching* (NABC for short). NABC facilitates reduction of data copies in web server's data sending path, by doing three things: (1) Layout the data in main memory in a way that protocol processing can be done without data copies (2) Keep a unified cache of data in kernel and ensure safe access to it by various processes and kernel and (3) Pass only the necessary meta data between processes so that bulk data handling time spent during IPC can be reduced. We realize NABC by implementing a set of system calls and an user library. The end product of the implementation is a set of APIs specifically designed for use by the web servers. We port an in house web server called SWEET, to NABC APIs and evaluate performance using a range of workloads both simulated and real. The results show a very impressive gain of 12% to 21% in throughput for static file serving and 1.6 to 4 times gain in throughput for lightweight dynamic content serving for a server using NABC APIs over the one using UNIX APIs.

1 Introduction

Increasing use of the Internet in various forms of business, communication and entertainment has placed an enormous performance demand on its key architectural elements: routers and switches at the middle of the net and web servers and proxy servers at the edge of the net. To get an idea of the magnitude of request burst a web site might be subjected to: it has been reported that the web site of 1998 winter Olympic games in Nagano, Japan has experienced a staggering 110,414 hits in a minute [11]. Similar numbers can be found for other popular web sites as well.

Web servers and proxy servers are extremely I/O intensive applications and it has been reported that a web server spends 70% to 75% of the whole processing

* The author carried out this work while being a full time Master's student of the Department of Computer Science and Automation, IISc.

S. Sahni et al. (Eds.) HiPC 2002, LNCS 2552, pp. 242–251, 2002.

time doing network and file I/O [10]. Therefore, optimizing the I/O processing for a web server is a possible means of scaling its performance.

It has been known that data copy is a bottleneck for high performance I/O [4] [2] [8]. Data copy results because of kernel's intervention in every I/O processing steps. In write/read calls, for example, data gets copied to/from the kernel from/to user applications. Apart from this, the mismatch which exists in the way various kernel subsystems handle data, results in copy while data is transferred from one subsystem to the other. For example, file system keeps data in page cache or in buffer cache while networking subsystem keeps data in fragmented form and needs header information for protocol processing. Therefore, a copy results while data is transferred from file system to the networking subsystem and vice versa.

In this paper we address this problem of data copy solely from the web server's perspective and propose a possible solution. Specifically we ask the question: How far, in a web server's data sending path, can we reduce data copies for both static and dynamic content processing? We address the problem without assuming presence of any special networking hardware.

In achieving the objective we propose a method of handling data which we call *Network Aware Buffering and Caching*, NABC. NABC facilitates layout of data in main memory, at the very first time it is generated, in such a way that protocol processing can be done without data copies (hence the rationale of the phrase *Network Aware Buffering*). NABC also facilitates in-kernel caching of outbound data and safe sharing of that data between processes and kernel. To achieve this, we implement a set of system calls in Linux-2.2.14 kernel and implement a set of C library interfaces for use by the web servers.

1.1 Organization

The rest of the paper is organized as follows. Section 2 discusses NABC in detail and discusses the design rationale behind the APIs. Section 3 outlines kernel and library implementation details. Section 4 presents the performance evaluation of NABC for a range of workloads. In section 5 we discuss related research efforts. Section 6 draws the conclusion.

2 Network Aware Buffering and Caching

2.1 Mechanisms

The main idea in NABC are the following:

- Layout the data in main memory at the very first time it is generated in such a way that protocol processing can be done without touching the data.
- Maintain a unified cache of outbound data in kernel and ensure that related processes and kernel can access it safely.

– When data is generated by one application and is sent out by another, then do not pass the whole data from one application to other by IPC, but pass only the portion needed to be examined and possibly modified (usually the header) and pass a reference to the bulk data body. Device a mechanism by which other application may send it by having a reference to the data body.

The first mechanism tries to reduce the need for data copy in protocol processing and requires that data be kept in fragmented form in memory, the second mechanism reduces the need for duplicate data buffering, and the overhead of data transfer from user application to kernel, and the third mechanism uses the first two and is a domain specific optimization for web servers so that bulk data transfer between processes is reduced.

2.2 The NC_handle ADT

In NABC `NC_handle` ADT effectively hides from the application programmer the intricacies of fragmented memory allocation and data placement. Figure 1 shows essential pointers in NC_handle ADT.

In NABC, a set of related data fragments are called *logical unit* of data. Data transfer from application to kernel takes place by remapping pages from process address space to kernel, this ensures data integrity in the cache. The processes trying to use a particular logical unit of data in the cache, specify that using an unique integer called *cache id*, associated with that unit. An access control mechanism limits access to these data units. NABC facilitates checksum caching on a per fragment basis. Precomputed checksum speeds up protocol processing when data is transferred from the cache.

2.3 NABC - APIs

The APIs can be logically divided into three categories: APIs for memory allocation and data buffering, APIs for cache management, and APIs for network

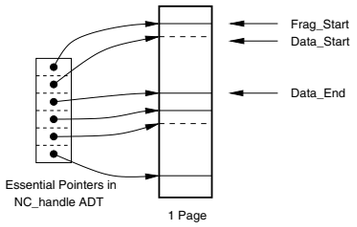

Fig. 1. Essential elements of `NC_handle` ADT. This simplified figure shows a page containing two fragments. There may be multiple pages in one `NC_handle` ADT and many more fragments. The pointers are generated by user library and the pages are mapped by a system call

```
int NC_alloc(NC_handle * handle, int size,
                              int fraglen);
int NC_read(int fd, NC_handle * handle,
        size_t handle_offset, size_t len);
int NC_insert(NC_handle * handle);
int NC_socksend(int sock_fd,
           size_t cache_id, size_t offset, size_t len);
```

Fig. 2. NC_alloc allocates memory in fragmented form, NC_read reads data to the to the fragments, NC_insert inserts data unit to the cache and NC_socksend sends data from the cache over network

transport initiation. Since the mechanisms proposed are targeted towards a particular kind of applications, the APIs are fairly small in number but rich in semantics.

APIs for memory allocation and data buffering let an application allocate memory in fragments and buffer data onto the fragments. Data may originate from a variety of sources.

The APIs for cache management give applications facilities for creating and destroying data units, collecting the statistics about the cache and controlling access to various data units. A process creating a data unit can give access to it to another process. This facilitates CGI processing where data generated by child process needs to be send out by parent.

Two APIs for sending data over socket are available in NABC. Due to space limitations all the APIs could not be shown. Full details of the APIs can be found in [17].

2.4 NABC and a Web Server

We envision that NABC will be useful for web servers which cache data in main memory for static content serving. In addition to caching static data, efforts are underway where dynamic content is also cached [5]. With dynamic content processing, we envision that the NABC scheme promises performance gain by substantially reducing copy of the data in its sending path.

The most straightforward situation is with static contents. We propose that header and data should be kept separately. That means all the information necessary for HTTP header generation is kept separately in the process address space and the bulk data is kept in the NABC cache and *cache id* is kept alongside the header information. So, when responding to a request, the header information is written separately using conventional `write` call and the data transfer is initiated using `NC_socksend` call.

CGI style dynamic content generation presents some problems for incorporating the scheme, so requires slight change in the way the CGI scripts are to be written. CGI/1.1 specifies meta variables and CGI extension headers. We use meta variables for passing the pid of the main server process to the child and

use the extension headers for passing the *cache id* and the data length from the child to the parent. Using these information, the parent can send the dynamic content to the client.

3 NABC – Implementation

NABC implementation was done on Linux-2.2.14 kernel. Apart from changes in kernel we have also implemented a user level C library.

3.1 Implementation Details

Kernel Implementation Details: Kernel implementation consists of implementation of a set of system calls. One set of calls map pinned down pages to process address space and unmap them. One set of calls do cache management and a third set of calls do network transport initiation. The NABC cache interact with the file cache when data is placed directly from file cache to the NABC buffers by copying. For all the data units that are placed in the cache, checksum of data fragments are calculated and cached, this speeds up protocol processing. The NABC cache interact with the networking code when the data is transferred over socket from NABC cache. In the kernel code, all the routines from `tcp_send_skb` and below in the protocol stack remain the same. One asynchronous thread does the job of a garbage collector.

Library Implementation Details: Library implementation is mainly responsible for making fragments out of mapped pages and creating the meta data about the fragments and keeping them in the `NC_handle` structure. When the applications make calls that inserts data into the NABC cache, this meta data is copied to a similar data structure in kernel.

3.2 NABC - Operation

When the modified Linux kernel is booted up, the data structure of NABC is allocated and the garbage collector thread is started as kernel daemon. The applications using the NABC, need to link a static library with their code.

4 Performance

Our experimental setup consisted of one PC acting as server machine with 500 MHz Pentium - III Processor having 320MB of RAM and four 100 Mbps network adaptors running our modified Linux2.2.14 kernel and four PCs acting as client machines each having Pentium - III processor, 64 MB of RAM and one network adaptor running Linux 2.2.18 kernel. All the machines were connected in a switched fast Ethernet.

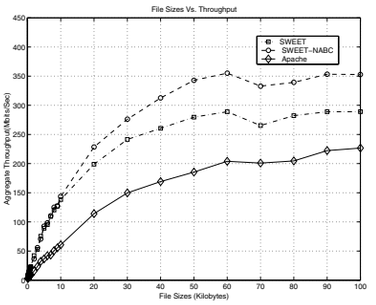

Fig. 3. Aggregate throughput Vs. file sizes when file sizes were varied from 100 Bytes to 100KB. The file sizes are varied in steps of 100 bytes below 1K, 1K between 1K and 10K, 10K between 10K and 100K. Aggregate throughput is the sum of the throughputs observed in individual machines

We have used an in-house web server called SWEET (*Scalable Web SErver using Events and Threads*) as our vehicle for performance evaluation. Very little change was necessary for porting SWEET to SWEET using NABC APIs (SWEET-NABC).

For dynamic data, we have used a scheme similar to fast CGI. But we did not implement fast CGI specification. In our method, a set of pre-forked processes does the dynamic data generation. On receiving the simple request for data, the server chooses a free process to generate the data. In our experiment on dynamic processing, the applications generate data from thin air and no real dynamic processing is involved. Since we wanted to expose the bottleneck the data copy imposes, this suffices our purpose.

As client simulation software, we have used an event driven program similar to *flashtest* available from Rice University [19].

4.1 Fixed Sized File Tests

In fixed sized file tests, a fixed number of files, *all* of same size, were fetched repeatedly in every experimental session. The number of files of a given size was taken as 100, 10 clients per machine was used and 10000 fetches per file size per machine was done. Figure 3 shows the plot of aggregate throughput vs. file sizes for file sizes between 100 Bytes and 100KB. Due to file caching, a file used to get served from the web server's cache except for the first time.

4.2 Modified SPECWeb99 Workload

We have used a modified version of SPECWeb99 guidelines for creating the static workload for evaluating the performance [1]. By doing this, we believe the workload we get is a faithful representative of real life workload.

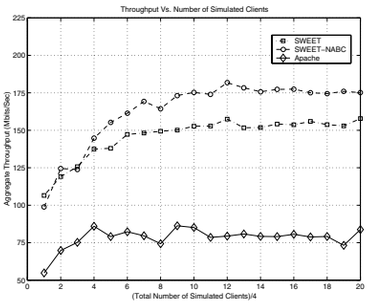

Fig. 4. Throughput Vs. Number of Clients for Modified SPECWeb99 Static Workload

Since we did not implement any cache replacement algorithm, total size of the data set that the server can serve at any experimental session is limited by the cache size that can be accommodated at once in physical memory, so we have kept the number of directories 25 across all experimental run. We have adhered to all the other guidelines of generating load and creating the files. 25 directories occupy around 125 MB of space.

Figure 4 shows change in throughput as the number of clients was varied. The total number of clients were varied from 4 to 80 in steps of 4. 1000 fetches (Different unrelated sequences per machine) per machine were used per experimental run. As can be seen SWEET-NABC almost always outperforms both Apache and SWEET. When the total number of clients are less, then both SWEET and SWEET-NABC operate at under capacity hence the I/O subsystems of OS is not stressed. As the number of simultaneous clients keep increasing, the server is compelled to serve more data. Therefore, data copying and checksum calculation becomes critical bottlenecks and SWEET-NABC with its in-kernel network aware data cache with pre-computed checksum starts to outperform SWEET.

4.3 Experiment With CSA WWW2 Trace

The CSA WWW2 server is used to serve the home page of about 160 students of Computer Science and Automation (CSA) department of IISc. In this experiment we took 1 weeks log of the server and performed the experiments on the subtraces of that trace based on the prefix cache size. The methodology is as follows: for a prefix cache size of X MB, we fed all the hits which fall within file set which contributed to that prefix cache size. The request was generated by a total of 64 simulated clients for 20 minutes from 4 client machines. we have tried to preserve the original request sequence.

Figure 5 shows the plot of the throughput vs. the prefix data set sizes. At 10MB of prefix size, SWEET-NABC outperforms SWEET by a margin of about 20%. At 50MB the gain is around 17% and at 100 MB it is around 9%. Two reasons contribute towards reduced performance advantage of SWEET-NABC

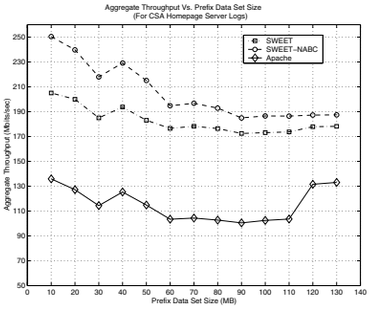

Fig. 5. Throughput Vs. Prefix Data Set Size for CSA WWW2 Server Logs

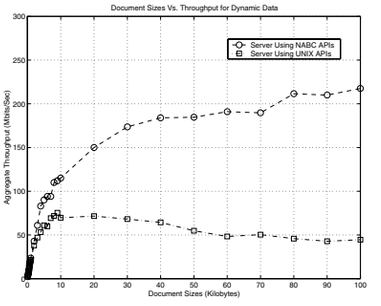

Fig. 6. Throughput Vs. Document Size for Dynamically Generated Documents

over SWEET with increased prefix data set size. Firstly, sweet does not use a very optimized cache lookup algorithm. With increase in prefix data set size, the number of files increase, and cache lookup time increases. Secondly, the file mix, and request mix follow a pattern in which the percentage of small files and request for them both are high [17]. With large number of small files, the per file overhead (connection establishment, teardown etc.) tends to overshadow the performance gain due to copy reduction (This is also evident in Figure 3 where appreciable performance gain is not there for small files).

4.4 Lightweight Dynamic Content Generation

The mechanism of dynamic content generation has been outlined before. We have varied the size of the documents being generated. 10 clients on each of 3 client machines were used to generate load.

As can be seen in Figure 6 that the performance gains in dynamic content processing is even more pronounced. Compared to POSIX APIs, in NABC scheme the number of copies reduced is more, hence we see a throughput gain of almost 4 times for 100KB file and 1.6 times when the document size is 10KB.

5 Related Work

A body of previous research efforts have focussed on reducing data copy for I/O, and enhancing IPC performance. Liedtke [14] discusses improvement of IPC performance in L3 μ kernel. Cranor and Parulker consider design and implementation of a new virtual memory system aiming at copy reduction [7]. Some previous research efforts have tried to solve the problem of copy free data transfer between user/kernel boundary in the context of I/O [4, 2, 3, 6, 18, 16]. All these systems have tried to export interfaces for general purpose applications. Some of the systems e.g., [6] provide same interfaces as POSIX, others like [2, 18, 16] require alternative non standard interfaces. The problems with these systems are that in many cases, the alternative interfaces are complicated to understand and use. For example, there is no parallel in POSIX of the concept of I/O-Lite context (`IOL_Context`) or I/O-Lite generator style traversal. Also, the mechanisms are not very efficient in all the situations. For example, in case of [6], the mechanism is dependent on the application behavior. In case of I/O-Lite, pathological conditions may arise in case of disk writes. Container Shipping [2] may end up using one page for transferring one packet.

More recent efforts in scaling web server have resulted in works like web server accelerator [13] which caches frequently accessed objects in a front node and uses a highly optimized protocol stack. Adaptive Fast Path Architecture (AFPA) [15] is a platform for implementing kernel-mode network servers on production operating systems. AFPA integrates the http processing with TCP/IP stack and uses a kernel managed zero-copy cache. In recent years, kHTTPd and TUX are two implementations of web servers in Linux kernel [12, 9]. Performance benefits notwithstanding, the problem with kernel mode web servers are the lack of fault isolation and lack of development support.

The path we are taking is a mixed one. Our aim is to facilitate the operation of user mode servers. We do this by keeping outbound data in kernel as much as possible so that copy is reduced in protocol processing and provide system call interfaces to the application processes so that the final control over the cached data lies with the application.

6 Conclusion

This paper discusses a new method of data buffering and caching for web servers called NABC. One immediate effort for evaluating NABC could be to evaluate how it performs against the other web server specific optimization efforts discussed in the previous section.

One point where NABC design lacks general appeal is its dependency on the nature of networking subsystem in Linux. But we believe that even if such constraints are removed the principle of NABC can be employed to keep a unified cache of outbound data. Although then file cache can be used as the the cache for static data, for dynamic data we may use the in kernel buffering as is done in NABC to reduce copy and processing time.

References

[1] SPECweb99 Release 1.02. `http://www.spec.org/osg/web99/docs/whitepaper.html`. 247

[2] E. Anderson. *Container Shipping: a Uniform Interface for Fast, Efficient, High-Bandwidth I/O*. PhD thesis, Computer Science and Engineering Department, University of California, San Diego, CA, 1995. 243, 250

[3] J. C. Brustoloni. Interoperation of copy avoidance in network and file i/o. In *INFOCOM (2)*, pages 534–542, 1999. 250

[4] J. C. Brustoloni and P. Steenkiste. Effects of Buffering Semantics on I/O Performance. In *2nd Symp. on Operating Systems Design and Implementation (OSDI)*, pages 277 – 291, Seattle, WA, Oct 1996. 243, 250

[5] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, NY, 1999. 245

[6] H. K. Jerry Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996. 250

[7] C. D. Cranor and G. M. Parulkar. The uvm virtual memory system. In *USENIX Annual Technical Conference*, Monterey, CA, June 1999. 250

[8] P. Druschel. *Operating System Support for High-Speed Networking*. PhD thesis, Department of Computer Science, The University of Arizona, 1994. 243

[9] Answers from planet TUX: Ingo Molnar responds. `http://slash-dot.org/articles/00/07/20/1440204.shtml`. 250

[10] J. C. Hu, S. Mungee, and D. C. Schmidt. Techniques for Developing and Measuring High-Performance Web Servers over high speed ATM. In *INFOCOM*, San Francisco, CA, 1998. 243

[11] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High-Performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2):17 – 26, Mar.-Apr. 2000. 242

[12] kHTTPd Linux http Accelerator. `http://www.fenrus.demon.nl`. 250

[13] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. M. Dias. Design and performance of a web server accelerator. In *INFOCOM (1)*, pages 135–143, 1999. 250

[14] Jochen Liedtke. Improving ipc by kernel design. In *14th Symposium on Operating Systems Principles (SOSP)*, Asheville, North Carolina, December 1993. 250

[15] R. King P. Joubert, R. Neves, M. Russinovich, and J. Tracey. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *USENIX Annual Technical Conference*, Boston, MA, June 2001. 250

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. I/O-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions On Computer Systems*, 18(1):37 – 66, 2000. 250

[17] S. Sen. Scaling the performance of web servers using a greedy data buffering and caching strategy. Master's thesis, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India, January 2002. 245, 249

[18] M. Thadani and Y. A. Khalidi. An efficient zero-copy i/o framework for unix. Technical Report SMLI TR95 -39, Sun Microsystems Lab, Inc., May 1995. 250

[19] Measuring the Capacity of a Web Server. Rice University. Department of Computer Science. `http://www.cs.rice.edu/CS/Systems/Web-measurement/`. 247