TCP/IP Cache Characterization in Commercial Server Workloads

Li Zhao, Ramesh Illikkal, Srihari Makineni
Communications Technology Lab
Intel Corporation

Laxmi Bhuyan
Department of Computer Science
University of California

{li.zhao, Ramesh.g.illikkal, srihari.makineni}@intel.com; bhuyan@cs.ucr.edu

Abstract – *Internet server applications (such as web servers, e-commerce front-ends, etc) spend a significant portion of time processing network data. These applications use TCP/IP as the communication protocol which is known to be very memory intensive. In this paper, we present a simulation-based characterization of the cache/memory access behavior of TCP/IP processing in two popular commercial benchmarks -- SPECweb99 and TPC-W. Our Simple Scalar simulator is fed with network traces collected from commercial platforms running these benchmarks under various configurations. We identify the types of data (descriptors, headers, control blocks, etc) that the TCP/IP stack needs to access while processing packets and analyze the cache behavior, in terms of cache size and locality for these data. We show that the TCP/IP data falls into two categories; data with temporal locality such as hash nodes and TCBs and data with no locality (transient) such as descriptors and payload. Based on the cache characterization study, we propose the usage of a dedicated cache to store and manage TCP/IP data with and without locality. We study various approaches to organizing the network cache and their effects. We show that a small cache, in the order of 5 Kbytes is sufficient for near-optimal performance of TCP/IP processing with the additional advantage of minimizing the processor cache pollution.. We also touch upon alternative approaches to enable network-friendly cache hierarchies without the need for a dedicated cache structure*

I. INTRODUCTION

TCP/IP [6] over Ethernet is the most dominant packet processing protocol in the data centers and on the Internet. Recent work [9,10] has shown that the networking requirements of commercial server workloads are significant. Specifically, it was found that the TCP/IP processing overhead in network intensive server benchmarks such as SPECweb99 and TPC-W can be 25% or higher. Hence it is important to understand the TCP/IP processing characteristics, uncover architectural issues and develop solutions to help reduce this processing overhead.

Figure 1 shows the current TCP/IP performance on the Intel® Xeon™ processor running Microsoft Windows* 2003 Server Enterprise Edition Operating System. For a 1460-byte application payload size (shown on x-axis), the TCP/IP stack was able to achieve only about 1 Gbps and that too with a 100% CPU utilization. An increase in available Ethernet network bandwidth (based on upcoming 10Gbps technologies) can not be utilized by applications unless the network processing overhead is alleviated.

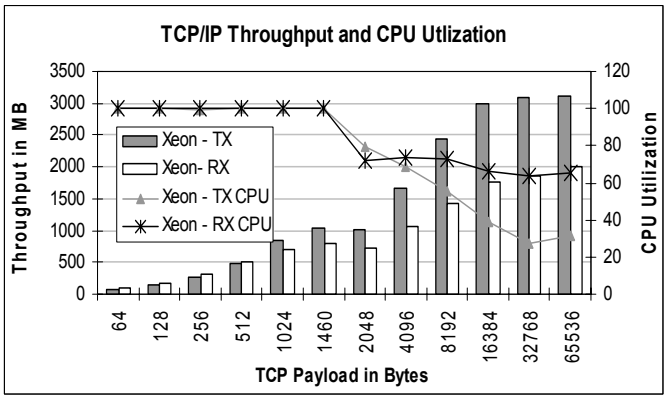

Figure 1. TCP/IP Performance

One of the major bottlenecks in TCP/IP processing has been shown to the significant number of memory accesses that are required on a per-packet basis. Our measurements on the XeonTM processor system have shown that for a 1460-byte payload size, the TCP/IP stack incurs approximately 20 and 63 cache misses for transmit and receive side processing respectively (despite having a 512KB L2 cache). These cache misses are due to accesses to various data types including network interface card (NIC) descriptors, TCP/IP header, TCP/IP control blocks and hash nodes. Another major source of cache misses is memory copy operations that are required to transfer data between the NIC buffers and the application buffers. In order to understand the nature of these memory accesses, our study presented in this paper focuses on (a) various types of data that TCP/IP stack needs to access during packet processing (b) cache size requirements and locality behavior for each data type and (c) better cache structures and policies to benefit network processing.

The contributions of this paper are as follows. By analyzing the locality properties of the data types associated with TCP/IP processing, we show that a small cache space (that is managed efficiently) is sufficient for TCP/IP processing. Based on this observation, we propose the notion of a network cache. The objectives of the network cache are (1) to minimize cache pollution by re-directing network data to a small dedicated cache and (2) to further partition the small dedicated cache into a locality-friendly cache (TLC) and a non-temporal stream buffer (SB).

II. OVERVIEW OF TCP/IP

In this section, we provide an overview of the TCP/IP receive and transmit side processing. TCP/IP processing begins when a packet is received by the Network Interface Card (NIC) or when the application sends a buffer down to the protocol stack for transmission. For these paths, we will now describe, at a high level, TCP/IP processing involved.

A. *Receive side Processing*

On the receive side, the processing begins when the NIC receives an Ethernet frame from the network and extracts the packet by removing the frame delineation bits. NIC then updates a data structure, called descriptor, with the packet information. TCP/IP stack supplies these descriptors to the NIC and are organized in circular rings. TCP/IP tells the NIC through these descriptors, among other things, where in the memory to copy the incoming packets. NIC then copies the incoming packet into a buffer in memory (via a DMA operation). Once the data is placed in memory, the NIC updates a status field in a data structure (known as a NIC descriptor) to inform the arrival of a received packet and generates an interrupt. This kicks off the receive side processing.

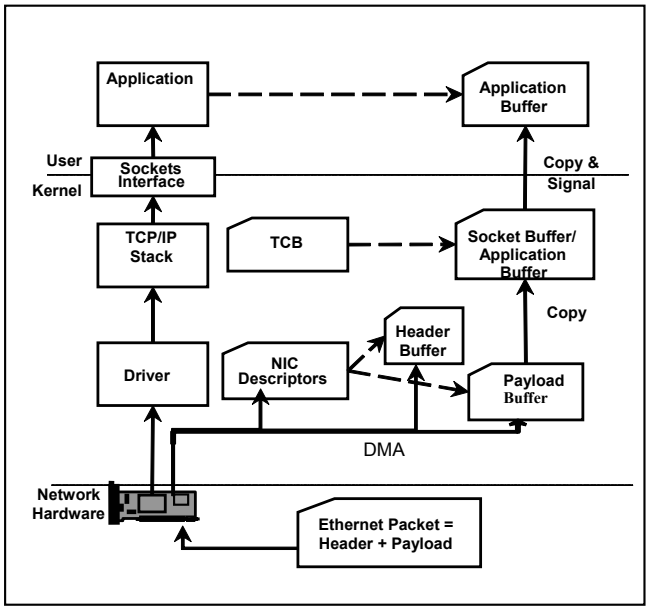

Figure 2. Data Flow in Receive-Side processing

Figure 2 shows the overall flow for receive-side processing. The NIC device driver reads the NIC descriptor to get packet header and payload information. This results in a compulsory memory access as the data will not be in cache (invalidated when the DMA occurred). Once the descriptor is read into the cache, the TCP/IP stack has access to the buffer containing the header and payload data. Accesses to the IP and TCP headers also end up as one or more compulsory memory accesses. The next step is to identify the connection that this packet belongs to. This information is maintained by the stack in a data structure called the TCP/IP Control Block (TCB). Usually the TCB lookup is based on a hash table with each hash key calculated from the IP address and port pairs. If there are several connections being processed simultaneously, there is a high probability that the TCB (and even the hash node) may not be in the cache, which forces another set of memory accesses. The data is then copied into an application buffer if one is already available in the TCB. Otherwise it is stored in a temporary buffer for later delivery to the application. The need for memory copy (or copies) from the payload buffer to the application buffer is one of the most time consuming operations in receive-side processing.

B. Transmit side processing

On the transmit-side, the processing starts when an application sends a buffer to the stack. The stack accesses the TCB associated with the connection and may then copy the application's data into an internal buffer. Optimizations to avoid this copy have been implemented in some stacks. When it is time to transmit data (based on the receiver TCP window size), the TCP/IP stack divides the accumulated data into Maximum Transfer Unit (MTU) segments. Typically, MTU is 536 bytes on the Internet and 1460 bytes on Ethernet LANs. It then computes the header (20 bytes for TCP assuming no options and 20 bytes for IP). These segments are then appended with the Ethernet header and passed down to the NIC driver. The driver sets up the DMA to transfer data to the NIC.

In this paper, we focus more on the receive side processing because it is known to be more memory intensive than transmit-side processing (especially due to the need for memory copies). However, it should be noted that most of our locality studies and cache enhancements would apply equally well to the transmit-side.

III. MEASUREMENTS METHODOLOGY

In this section we provide details on the commercial workloads used in this study, how we have extracted network traces from these commercial workloads and the

simulation environment we use in this paper to study the cache characteristics of TCP/IP processing.

A. Network-Intensive Commercial Workloads

In order to capture the typical scenarios in a data center, we start by looking at popular industry benchmarks. We chose TPC-W and SpecWeb99 because they represent workloads that are network intensive. To represent web server workloads, we chose SPECweb99 since it is a benchmark that mimics the load (70% static and 30% dynamic requests) in a web-server environment. The performance is measured in terms of the number of simultaneous connections that the web server can support. A primary requirement for SPECweb99 is that the network rate needs to be within 320 Kbps and 400 Kbps per connection. The average file size transmitted (per HTTP operation) in SPECweb99 is around 14KB.

To understand the performance of front-end servers in e-commerce environments, we chose the TPC-W benchmark. The TPC-W benchmark is modeled after an online book store. TPC-W specifies 14 different types of web interactions that require different amounts of processing on the system(s) under test (SUT). Each web interaction requires multiple requests and responses (for the HTML content, the images, and other data) between the clients, the web server, the image server, the cache server and the back-end database server. The average size transmitted to the client per web interaction is 46KB. In this paper, we focus on the three network-intensive servers in the setup; namely the web server, cache server and image server.

B. Network Traffic Characteristics and Tracing

To understand the network traffic in these two commercial workloads, we collected network traces using an Ethernet packet sniffer [11] while running these benchmarks.

Figure 3. represents the packet distribution in the server workloads we studied. The packet sizes and distribution varies considerably between the workloads. As shown in the figure, roughly 20 to 40% of the packets are control packets (containing zero-length payloads). Except for the cache server, we find that other workloads exhibit a small fraction of packets that have a payload size of under 1400 bytes. For these servers, almost 60% of the packets have a payload size that is very close to the maximum application payload size allowed in Ethernet frames, which is 1460 bytes. The cache server shows payload sizes around 150 bytes and between 1200 and 1460 bytes.

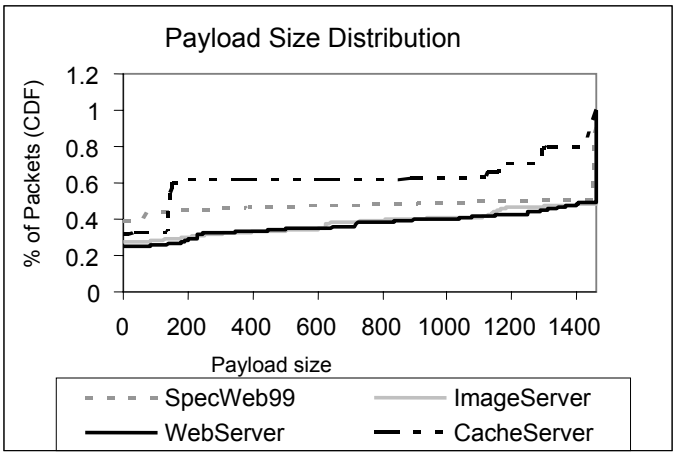

Figure 3. TCP/IP Payload Size distribution

We also analyzed the network traces from these benchmark applications to figure out the number of back to back packets for the same connection, which we call train of packets. This train of packets forms the basis for our locality study. Later we show that larger number of back to back packets for the same connection improves the locality exhibited by some of the TCP/IP data types. A graph with the number of packets trains against the number of packets in the train is charted below.

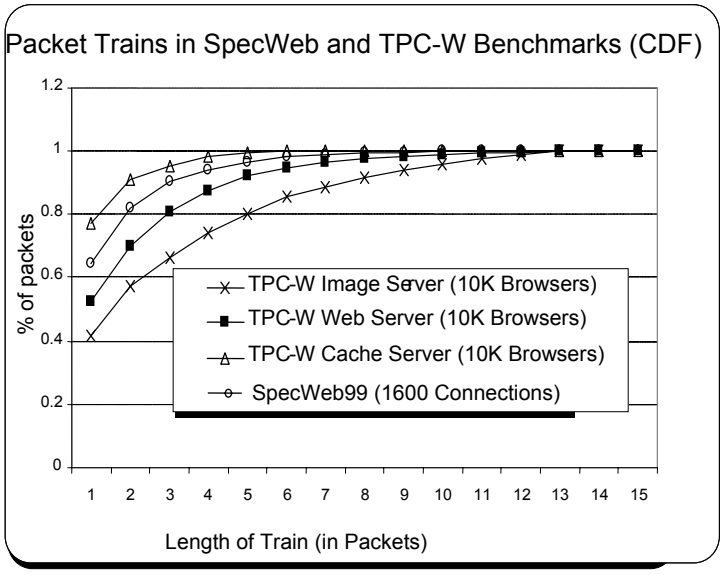

Figure 4: Packet Trains in TPC-W

As shown in the Figure 4, all the workloads have packet trains, although TPC-W image and web servers have more packet trains than the rest.

C. *Simulation Methodology*

To understand the cache behavior of TCP/IP processing, we used an execution-driven simulation methodology based on the SimpleScalar simulator [12]. Our base system configuration is a four-way fetch/issue/commit MIPS micro-processor with an instruction window size of 128 entries, 2 integer units, 2 load / store units and a floating point unit. We simulate a two-level cache hierarchy. The L1 Icache and Dcache configuration are 32 Kbytes in size, contain 64-byte cache lines and are 4-way set-associative. The data cache is write-back, write-allocate, and non-blocking with 2 ports. The L2 is a unified, 8-way, 1Mbyte cache with 64-byte cache lines and a 15-cycle cache hit latency. The main memory latency in the base configuration is 300 cycles. We ran our highly optimized TCP/IP stack in SimpleScalar in execution driven mode. The stack was fed with the network traces captured from the SPECWeb99 and TPC-W benchmarks.

We also modified the cache subsystem by adding a dedicated network cache (details in section V) to study the caching behavior of the network component of the benchmarks under study.

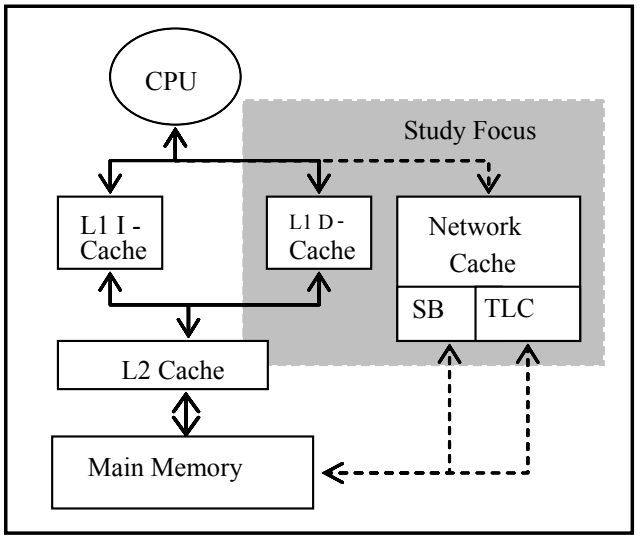

Figure 5. Network Cache Configuration

As shown in Figure 5, we implemented a dedicated cache that caches a subset of the network data (these bypass the existing cache hierarchy). Furthermore, we also studied two different cache structures for the network cache – a stream buffer structure and a locality-friendly structure that will be explained in detail in a later section.

IV. DATA CACHE BEHAVIOR OF TCP/IP

In this section, we discuss the locality behavior, cache size requirements and the impact of number of simultaneous connections on each of the TCP/IP data types. We categorize these data types into two categories – the ones which exhibit locality and the ones which don't. Figure 6 shows all the data types touched by TCP/IP processing.

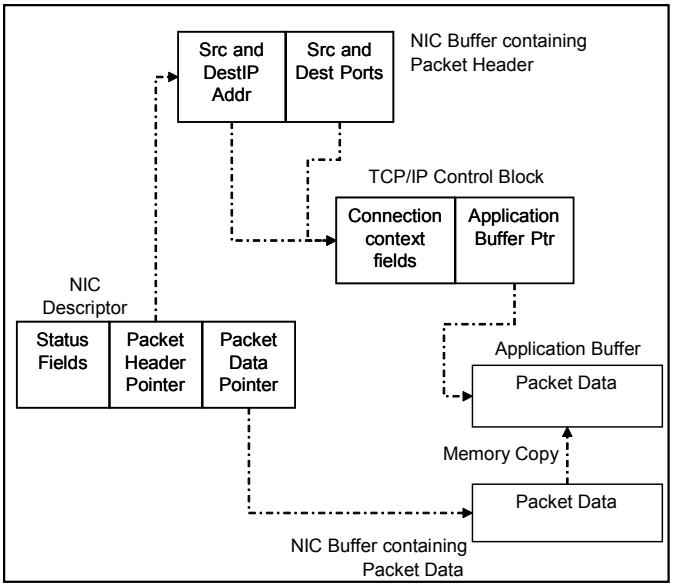

Figure 6. Memory Accesses in TCP/IP Processing

A. *Descriptors*

As explained before, NIC descriptors are data structures used by TCP/IP stack to communicate with the NIC. The stack maintains two sets of descriptors; one for transmit side and one for receive side. Typically, a TCP/IP stack allocates some number of descriptors (256 for instance) for each side and places them in a circular queue. The size of a descriptor is small (typically within a cache line; 64 bytes in our case). When the NIC receives packets, it updates the fields in the receive descriptor with the information related to the arrived packet before copying the descriptor and the payload into the memory using Direct Memory Access (DMA). To maintain cache coherence, all associated copies of the descriptor in the processor cache get invalidated. The stack then reads the descriptor and starts processing the packet. This read is bound to result in a cache miss since it was invalidated. In case of transmit, the TCP/IP stack fills the transmit descriptor fields appropriately, sets up the DMA engine to copy the data from the application or TCP/IP buffer space to the NIC device. Upon transmitting the data, the NIC will update the descriptors. Similar to the receive-

side, the TCP/IP stack will incur one or more misses when accessing the descriptors.

In summary, the descriptors don't show significant cache locality. Some modern NICs can interrupt once for several packets rather than for every packet. In such scenarios, a larger cache line may help since if the descriptors are placed in contiguous memory locations and are reused all the time.

B. TCP/IP Header

The NIC descriptor provides information regarding the incoming packet header and payload. Because the header is also new data placed into system memory by the NIC device, this data will not show any cache locality either. So, when the stack needs the header data, the processor has to fetch it from the memory. For a TCP/IP packet without any TCP options turned on, the header size is 40 bytes. In addition, in our study, we used header splitting -- this requires the NIC to split the header and the payload and copy these into two separate buffers provided by the stack in the descriptors. This feature could be added to future NICs to achieve cache line and page boundary alignments.

C. Payload

The payload represents the application data in a TCP/IP packet (essentially without the TCP/IP protocol header). For a regular Ethernet frame size of 1514 bytes, the size of the payload without the header information is 1460 bytes. Just like the header and the descriptor, the payload is also new data coming into the system and hence does not show temporal re-use in the cache. But this data shows spatial locality (larger lines helpful) as it occupies contiguous space in the memory. Prefetch engines in the processors can take advantage of this (especially when a copy operation is performed).

D. TCP/IP Control Block (TCB)

TCB is a data structure that the TCP/IP stack uses to store connection context information. The TCB size in our TCP/IP stack is 512 bytes. However, the stack typically only touches a portion of the TCB depending on what is happening on that connection. i.e., transmitting, receiving, sending acknowledgements, etc. We expect the TCB data structure to have good cache locality since packets belonging to the same connection access the same TCB. To understand the impact of the number of simultaneous connections on TCB accesses, we measured the TCB miss ratio for TPC-W Image Server at varying number of Emulated browsers with 8 Kbytes of cache space to store the TCBs. To accomplish this, we modified the simulator by adding a separate 8 Kbytes of cache for TCBs and redirected any TCB accesses to this cache. The graph in Figure 7 shows the simulation results. As expected, the TCB miss ratio increases as the number of simultaneous connections increase.

To understand the TCB locality, we measured the miss ratio of TCBs as a function of the cache size (for a fixed number of simultaneous connections). Figure 8 shows the results from our simulation for SPECweb99 and TPC-W servers. It can be observed from the graph that the miss ratio reduces as the cache size is increased. However, we also find that the reduction in miss ratio slows down beyond 4 Kbytes (for 3 of the four server workloads). This shows that the TCBs exhibit cache locality across multiple packets. This locality is quite dependent on the number of back-to-back packets processed for a connection.

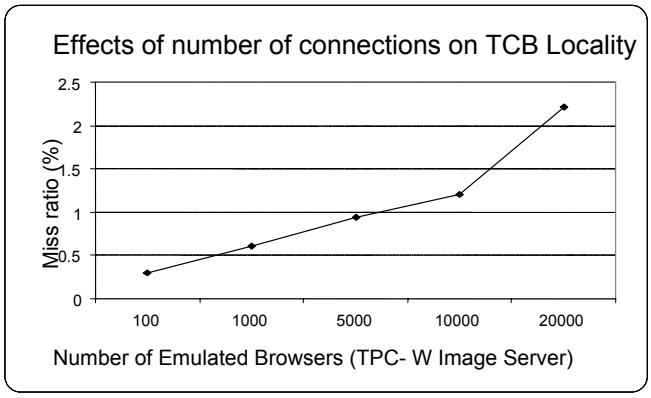

Figure 7. TCB miss ratio vs. connections

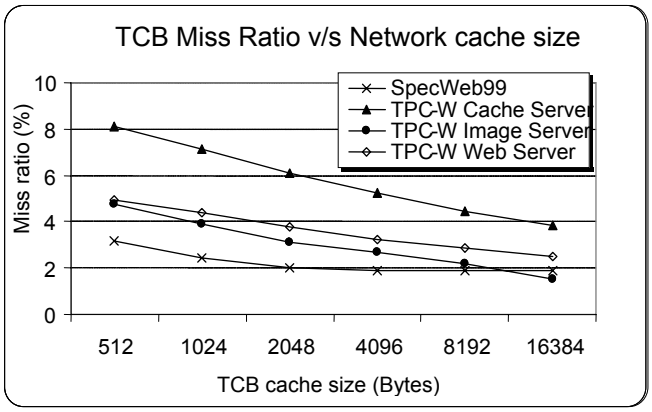

Figure 8. TCB miss ratio at various cache sizes

E. Hash table

Many server applications (such as web, ftp and mail) typically handle thousands of simultaneous connections

(each requiring a TCB data structure). In order to look up a TCB for a connection quickly, TCP/IP stacks typically employ hashing. The hash key is calculated by using the source and destination IP addresses and port numbers. There have been several studies [13] on how to achieve faster lookups. The FreeBSD stack (for instance) uses a hash table with each entry (hash node) in the table pointing to a linked list. This linked list is used to store TCBs that fall in the same hash node. To take advantage of back-to-back packet arrivals on the same connection and to avoid the traversal of the linked list multiple times for the same TCB, the FreeBSD TCP/IP stack moves recently accessed TCBs to the front of the linked list.

Figure 9 shows the hash node miss ratio as a function of the cache size (for a fixed number of connections). We have measured this by simulating a dedicated cache for storing the hash nodes and modifying the SimpleScalar simulator to redirect any accesses to the hash nodes to this new cache. As expected, the behavior looks similar to that of the TCB miss ratio shown in Figure 8.

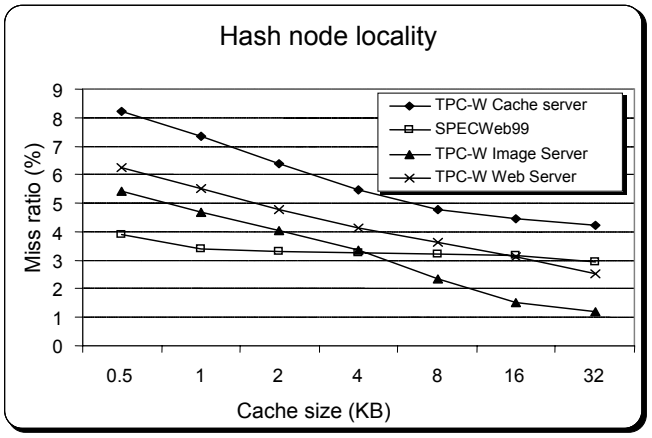

Figure 9. Hash node locality

F. TCP/IP Stack Variables

Apart from the data types described earlier, the stack also uses local variables (like any other program). These also have temporal locality because the same code and local variables gets used for each data packet. Since the size of this data is relatively small and is independent of the number of simultaneous open connections, we didn't perform any specific studies to quantify the locality of this data type.

In summary, of the TCP/IP data types, the TCBs, Hash nodes and the local variables exhibit high degree of locality. NIC descriptors, Headers and Payload have minimum or no locality. We will use this observation in the next section to reduce the cache size requirements of a "network" cache.

V. NETWORK CACHE

Our next step was to study the cache pollution effects and to derive an optimal cache organization for TCP/IP. In order to accomplish this, we simulated a dedicated Network Cache (NC) for TCP/IP in SimpleScalar. The motivation behind using a dedicated cache for TCP/IP is the following: (1) to reduce the cache pollution caused by the transient network data on application cache, (2) to eliminate the effects of application interference on network-related data and thereby improve the TCP/IP performance, and (3) to provide flexibility in organizing and managing the cache in a way that benefits the TCP/IP performance. It should be noted that this study is especially useful in the context of TCP/IP offload solutions [3,5,7] where consumption of additional die space is precious. In this section we study the benefits of using the network cache with various cache configurations.

A. Design and Organization

TCP/IP data, from a cache behavior point of view, falls into two categories – data that is cache-friendly (exhibits locality) and the data that is transient (mostly touched once). So, we divide the network cache into two parts:

- *TCP Locality Cache (TLC):* This cache stores data that exhibits locality, i.e. TCBs, hash nodes and local variables. As a result, a typical cache organization (good associativity, line size, etc) will help.
- *Stream Buffer (SB):* This buffer stores transient data, i.e. NIC descriptors, packet headers and payload. We start by organizing this as a FIFO buffer. It should be noted that the size of each entry in the buffer is still a cache line (in order to exploit spatial locality at least within the line).

In this paper, we concentrate on the characterization of the network cache. One implementation approach is to create a new memory region for network-related data. The details of this and other implementation approaches are not within the scope of this paper.

B. Evaluation of Network Cache

We study the impact that that network cache (TLC and SB) has on TCP/IP cache performance in the following subsections.

1) TLC Performance

We modified SimpleScalar to detect all accesses to cache-friendly data (TCBs, hash nodes, local variables)

and redirected these to the TLC. We varied the size of the TLC to study the number of misses per packet for different workloads. Figure 10 shows the results. From the graph, we can see that, for all the workloads, the misses per packet reduces significantly up to a TLC size of 4 or 8 KB. Beyond this size, the rate of decrease in misses per packet is minimal. Beyond 8 KB, the misses per packet remains almost constant. . This graph also shows that even though the data stored in TLC exhibits temporal locality, number of misses per packet for a given cache size is not the same across different workloads because each has different number of simultaneous open connections and the number of back-to-back packets.

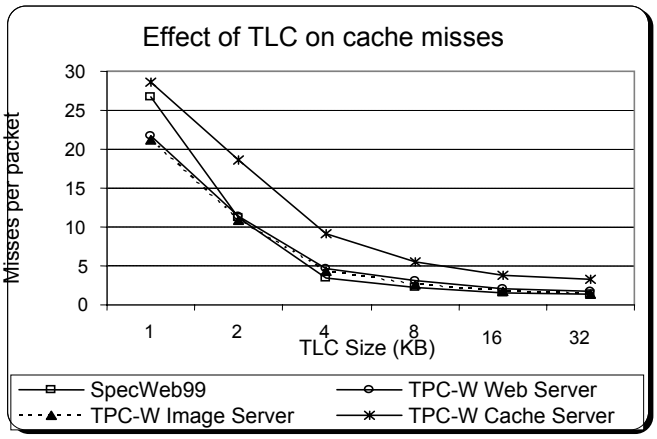

Figure 10. Impact of TLC on misses

2) Stream Buffer Performance

To simulate a FIFO stream buffer (SB), we employed a fully associative cache with FIFO replacement policy in SimpleScalar. We modified the simulator to detect all accesses to non temporal data (descriptors, payload and headers) and redirected these to the SB. We measured the number of cache misses per packet caused by this non temporal data for varying SB sizes. The results are shown in Figure 11. From the figure, it is clear that a small stream buffer of one to two Kbytes is all that is required by TCP/IP non temporal data. We can reduce the size of the SB further by using a small amount of area (say, 10 cache lines) for payload and streaming the application data through this when we are copying the payload between the application and TCP/IP stack. We can reuse this area during the copy operation because TCP/IP stack doesn't touch the data bytes once they are copied.

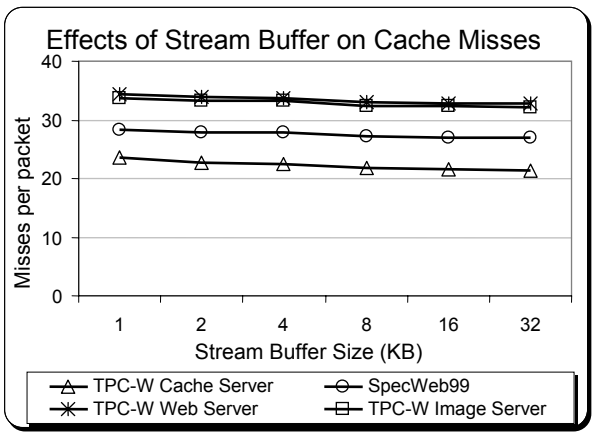

Figure 11. Impact of Stream Buffer on misses

3) Network Cache - Combined vs. Split

Here, we study the effectiveness of our proposed split NC organization (TLC + SB) against a combined cache organization in terms of the number of cache misses per packet. We already showed in Figure 11 that having a larger SB does not improve the misses per packet. So, we fixed the SB at 1 Kbytes and varied TLC size. We measured numbers for SPECweb99, TPC-W web, image and cache server workloads.

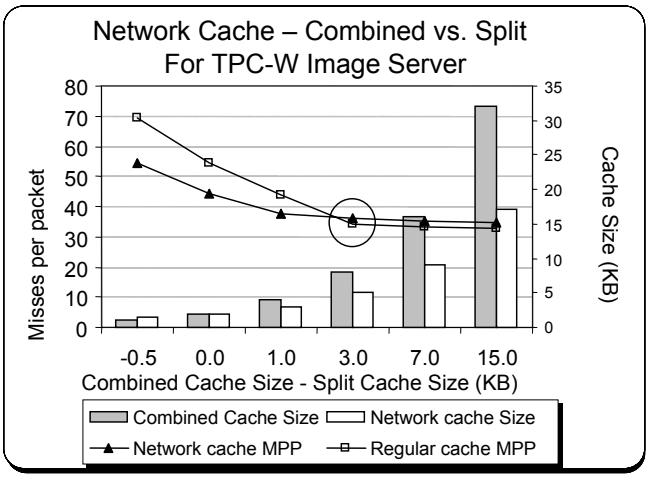

Figure 12. Combined Cache vs. Split Cache

For brevity, we show the results only for the TPC-W Image server in Figure 12. However, we have observed same behavior across all the workloads. The x-axis in Figure 12 shows the cache size difference between the combined cache and network cache (in KB). On the first y-axis (left) of Figure 12 shows measured cache misses per packet and the second w-axis (right) shows the absolute cache sizes. To achieve a miss per packet of 34 (circled data point in the graph) we need a combined cache size of 8 Kbytes vs. 5 Kbytes for NC. This is mainly because we eliminate the chances of transient data replacing the temporal data by separating the cache into TLC and SB. Since there is no such separation in

case of combined cache, we can not avoid these untoward evictions. If this combined cache is just part of the regular L1/L2 cache, then the effect of application pollution further deteriorate its performance.

VI. Alternate Approaches and Opportunities

The approach studied in this paper was that of a small dedicated cache for network processing. Here we discuss other potential approaches as well as potential optimizations to the network cache approach. Optimizations to the network cache approach can be thought of when considering the use of other techniques such as data prefetching and forwarding [14, 15]. For example, when designing a stream buffer, an optimal balance has to be achieved between its size and the amount of prefetching allowed. One other extension of SB usage is that the NIC can push the payload directly into the SB without incurring pollution in the regular cache. Additionally, cache line locking schemes can be used to make sure that the pushed data stays until the TCP/IP stack consumes it. This is not easy to guarantee with the general purpose caches without affecting other running applications.

When dedicating a small amount of cache for TCP/IP processing is not feasible, an alternative approach is to proactively invalidate cache lines [16, 17]. For example, all of the transient data that are redirected to the stream buffer can be invalidated after its use. In particular, it is important to invalidate the payload (at least the source) after the buffer copy is complete. Another possibility is to invalidate the TCB data structure when the connection is closed. We are currently modifying the stack to issue such invalidates for the payload and the TCBs.

VII. Conclusions and Future Work

In this paper, we have studied the cache behavior exhibited by various types of TCP/IP data. We have shown that the descriptors, header and application data are transient data and hence do not have any temporal locality. On the other hand, TCBs and hash nodes show temporal locality because at any point of time there are one or more back to back packets for the same connection for popular commercial workloads like SPECweb99 and TPC-W. Based on this initial observation, we proposed a dedicated cache called a Network Cache (NC) for storing TCP/IP data. Our network cache organization was based on two different structures (TLC and SB). We have shown that 1 to 2 Kbytes of SB is enough for storing the transient data. We have also shown that it took 8 Kbytes of combined cache versus 5 Kbytes of network cache (SB = 1 Kbytes, TLC = 4 Kbytes) to achieve the same cache performance.

Future work involves expanding the study to cover other popular network intensive commercial workloads. We also plan to extend the simulation environment to study the impact of cache pollution caused by the TCP/IP transient data and the benefits of early invalidation of cache lines in the context of real applications. We also like to explore further optimizations that can be applied to cache such as line locking, using a small area of stream buffer to stream the payload data during copy operations and using prefetch mechanisms to prefetch data into the TLC and SB.

References

[1] J. Chase et. al., “End System Optimizations for High-Speed TCP”, IEEE Communications, Special Issue on High-Speed TCP, June 2000.
[2] D. Clark et. al., “An analysis of TCP Processing overhead”, IEEE Communications, June 1989.
[3] A. Earls, “TCP Offload Engines Finally Arrive”, Storage Magazine, March 2002.
[4] A. Foong et al., “TCP Performance Analysis Re-visited,” IEEE Int’l Symposium on Perf. Analysis of Software & Systems, Mar 2003.
[5] K. Kant, “TCP offload performance for front-end servers,” to appear in Globecom, San Francisco, 2003.
[6] J. B. Postel, “Transmission Control Protocol”, RFC 793, Information Sciences Institute, Sept. 1981.
[7] M. Rangarajan et al., “TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance,” Rutgers University, Technical Report, DCS-TR-481, March 2002.
[8] G. Regnier et al., “ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine,” HOT Interconnects, 2003.
[9] S. Makineni and R. Iyer, “Performance Characterization of TCP/IP Processing in Commercial Server Workloads”, to appear in the 6th IEEE Workshop on Workload Characterization (WWC-6), Oct 2003.
[10] S. Makineni and R. Iyer, “Architectural Characterization of TCP/IP Packet Processing on the Pentium® M microprocessor”, to appear in the 10th High Performance Computer Architecture, Madrid, Feb 2004.
[11] Finisar Systems, http://www.finisar.com/
[12] SimpleScalar LLC, http://www.simplescalar.com
[13] P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming tcp packets,” SIGCOMM '92, Aug. 1993.
[14] D. Koufaty, X. Chen, D.K. Poulsen and Josep Torrellas, "Data Forwarding in Scalable Shared-Memory Multiprocessors," IEEE Transactions on Parallel and Distributed Systems, Dec 1996.
[15] D. Koufaty and J. Torrellas, "Comparing Data Forwarding and Prefetching for Communication-Induced Misses in Shared-Memory MPs," Int’l Conference on Supercomputing, July 1995.
[16] A. Lebeck, D.A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," Proc. of the 22nd Annual Int'l Symp. on Computer Architecture, 1995.
[17] A. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," ACM/IEEE International Symposium on Computer Architecture (ISCA), May 2000.