A New TCB Cache to Efficiently Manage TCP Sessions for Web Servers

Guangdeng Liao[1], Laxmi Bhuyan[1], Wei Wu[2], Heeyeol Yu[3], Steve R. King[2]
[1]University of California, Riverside, USA
[2]Intel Corporation
[3]Cisco Systems, Inc
{gliao, bhuyan}@cs.ucr.edu {wei.a.wu, steven.r.king}@intel.com heeyu@cisco.com

ABSTRACT

TCP/IP, the most commonly used network protocol, consumes a significant portion of time in Internet servers. While a wide spectrum of studies has been done to reduce its processing overhead such as TOE and Direct Cache Access, most of them did studies solely from the per-packet perspective and concentrated on the packet memory access overhead. They ignored per-session data TCP Control Block (TCB), which poses a challenge in web servers with a large volume of concurrent sessions.

In this paper, we start with challenge studies and show that the TCB data should be efficiently managed. We propose a new TCB cache addressed by session identifiers to address the challenge. We carefully design the TCB cache along two important axes: cache indexing and cache replacement policies. First, we study the performance of various hash functions and propose a new indexing scheme for the TCB cache by employing two *Universal* hash functions. We analyze session identifiers and choose some important bits as indexing bits to reduce hashing hardware complexity. Second, by leveraging characteristics of web sessions, we design a *speculative* cache replacement policy, which can effectively work on the TCB cache with two cache banks. Experimental results show that the new cache efficiently manages the per-session data. When it is used in TOEs or integrated into CPUs to manage the per-session data, TCP/IP processing time is significantly reduced, thus saving web server response time.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: *Network communication.*

General Terms

Design, Performance.

Keywords

TCB, Cache, TOE, Web Servers, Hash Functions.

1. INTRODUCTION

TCP/IP over Ethernet is the most dominant communication protocol in commercial servers such as web server, e-commerce, database, storage over IP, etc. Existing research [3, 10, 11, 14, 17, 18, 20, 32] has shown that TCP/IP packet processing, especially in the receive side, consumes a significant portion of time in those servers. Specifically, it was found that the TCP/IP processing overhead in high performance web servers such as Flash web servers can reach up to 80% of the time [14], and the processing in the receive side over 10 Gigabit Ethernet network (10GbE) easily saturates two cores of an Intel Xeon Quad-core processor [17]. Hence, it is important to optimize TCP/IP packet processing so that Internet servers can ultimately provide better web service.

A wide spectrum of architectural research has been conducted for TCP/IP to optimize its processing performance [3, 6, 9, 10, 11, 17, 21, 22, 29]. They fall into two categories: Offloading protocol stack into NICs like TCP Offload Engine (TOE) [6, 9, 22, 29] and pushing NICs closer to CPUs such as Direct Cache Access (DCA) or integrated NICs [3, 10, 11, 17]. TOE offloads the whole protocol stack to eliminate the processing overhead. In contrast to TOEs, Intel proposed DCA to route network data into CPU caches for eliminating the packet memory access overhead [9]. Without modifying the TCP/IP protocol stack, Binkert *et al.* [3] integrated a simplified NIC into CPUs to naturally implement DCA and to eliminate long latency access to device registers.

However, previous studies were conducted from the per-packet perspective and focused on the packet memory access overhead. They paid no attention to per-session data TCP Control Block (TCB). TCB is a per-session data structure of 512 bytes that TCP/IP uses to store its TCP session states and is accessed on the TCP critical path [4, 9, 14, 24]. A large number of sessions and web session behavior in web servers complicate the management of TCBs and introduce challenges: 1) in TOEs, TCB is accessed in the critical path and protocol processing stalls until the data is ready [9]. Hence, TCP/IP processing performance heavily relies on how fast TCB is accessed. Typically, TOEs put a CPU-like dedicated cache associated with modular indexing and the Least-Recently-Used replacement policy (LRU) to manage the per-session data. Unfortunately, thousands of concurrent web sessions substantially increase cache design complexity and those conventional cache designs are inefficient; 2) besides TOEs, the TCB access overhead is also significant in web servers while running the protocol stack on CPUs [14]. That is because a large number of web sessions enlarge the working set size and incurs cache pollution, and long intervals between two page requests (or *user thinking time*) in the same session increase the reuse distance of the per-session data deteriorating cache pollution. The above challenges motivate us to design a new dedicated TCB cache to manage the per-session data.

In this paper, we design a new TCB cache with extensive consideration of web session characteristics. The proposed TCB cache is addressed by the session identifier, contained in the header of the incoming web request. To provide high cache performance, we design the cache along two important axes: cache indexing and cache replacement policies. We observe that the traditional modular hash is not a perfect fit for the TCB cache due to its uneven cache accesses. Prior studies on CPU caches [13, 25, 31] have demonstrated that XOR-based hash and Prime-based hash can reduce cache conflict misses for SPEC CPU benchmark applications. Motivated by these studies, we evaluate the performance of various hash functions and propose a new cache indexing scheme for the TCB cache by employing multiple *Universal* hash functions [5]. In hash literature, multiple *Universal* hash functions have been analyzed and confirmed that they can lead to a more even distribution of load across hash buckets [5]. Our studies reveal that two cache banks based on *Universal* hash functions for a TCB cache perform well. Since all session identifiers share the same destination port and IP address for a web server, we reduce a session identifier from a 4-tuple of 96 bits (src IP, src port, dst IP, dst port) to a 2-tuple of 48 bits (src IP, src port). To further reduce hashing hardware complexity, we do a bit-by-bit analysis and choose 16 important bits as indexing bits for the TCB cache. Using the tailored indexing bits reduces the hashing hardware complexity by a factor of 3 while retaining the same cache performance.

Although multiple cache banks can achieve an even cache access distribution, they make it difficult to implement cache replacement policies in hardware [13, 25, 26, 27, 31] and sacrifice the effectiveness. In this paper, we design a *speculative* cache replacement policy to resolve the above issues by leveraging web session characteristics. Each session in web servers exhibits an *ON/OFF* model, where the periods during the file transfer and the idle times are referred to as the *ON* period and the *OFF* period, respectively [2, 8]. During the *ON* period, the corresponding TCB is frequently accessed, whereas no TCB access occurs during the *OFF* period. *ON* and *OFF* periods interleave in each session. In the *speculative* replacement policy, we predict TCB blocks with the *ON/OFF* status and aim at keeping *ON* cache lines as long as possible. In addition, we propose migrating the replaced *ON* cache lines to another cache set with *OFF* cache lines based on auxiliary *Universal* hashing. We perform a detailed hardware design to show that the above policy can be implemented at a reasonable hardware cost and outperform existing cache replacement policies on multiple cache banks [13, 31].

In order to evaluate cache designs, we developed a trace-driven cache simulator and experimented with four real web server traces: Boston University trace (BU), NASA-HTTP (NASA), ClarkNet-HTTP (Clarknet), Saskatchewan-HTTP (Sak). Simulation results show that the new TCB cache achieves much lower miss ratios than the original TCB cache. When it is used in TOEs or integrated into CPUs to manage the per-session TCB data, we can significantly reduce TCP/IP processing time, thus saving web server response time.

The remainder of this paper is organized as follows. The next section describes our preliminary studies to motivate the research. The designs of the new TCB cache architecture are elaborated in Section 3 and experimental results are presented in Section 4. Finally we discuss related work and conclude the paper in Section 5 and 6, respectively.

2. PRELIMINARY STUDIES

2.1 TCB Challenges

A wide spectrum of optimizations has been done for TCP/IP to improve its processing performance. They fall into two categories: offloading the protocol stack into NICs (TOE) [6, 9, 22, 29] or pushing NICs closer to CPUs while keeping protocol processing on CPUs [3, 10, 18, 21] such as DCA or integrated NIC etc. In this subsection, we study challenges on these two schemes from a large number of sessions in web servers to motivate our research.

2.1.1 Challenge in TOEs

Intel presented its 10Gbps TOE's detailed designs in [9] and the major function units are illustrated in Figure 1a. Input sequencer analyzes an incoming packet and extracts the 4-tuple session identifier from the packet header. The packet is stored into memory sitting on-board or connected externally for future transfer to applications. The session to which the packet belongs is looked up and the session data is loaded into internal working registers used by the execution unit. Then, the execution unit, controlled by instructions from the instruction ROM, performs the central part of the protocol processing using the session data. The complete micro-program implemented to perform TCP inbound processing consists of ~300 lines of code. The TCP fast path processing for in-order packets in a session takes 116 instructions and the slow path processing with complex out-of-order control have ~300 instructions. In most of the cases, incoming packets are in-order and thus belong to the fast path.

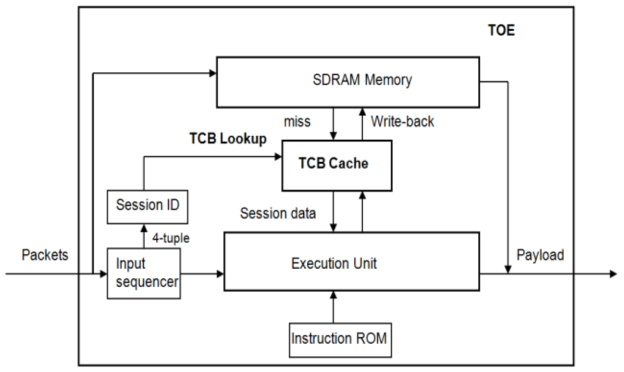

Figure 1a. Function units in TOEs

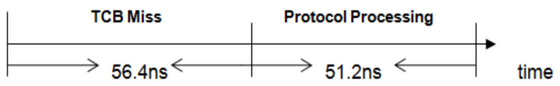

Figure 1b. Processing time with a TCB miss

In TOEs, TCB is accessed before protocol processing and the processing stalls until the data is ready. The data is returned from the TCB cache with a cache hit, otherwise, it is fetched from the memory. It was reported in [9] that 51.2 *ns* is required for in-order packet protocol processing in a 10Gbps TOE. With a TCB cache miss, Figure 1b shows the overall packet processing time, where we assume that memory access latency is 50 *ns* and each cache miss incurs only one memory access (TCBs are typically organized by a hash table in the memory and the TCB entry is found by traversing a linked list in each hash table bucket [4]. A TCB cache miss incurs both the linked-list traversal and data accesses, thus causing more than one memory accesses). Fig.1b reveals that TCB access takes more than 50% percent of the overall processing time and much higher if we consider several memory accesses for a cache miss. With a cache hit, the TCB access latency can be substantially reduced to 6.4 *ns* [9]. Hence, the packet processing performance heavily relies on how fast TCB is accessed. Currently, the TCB cache is implemented as a traditional cache associated with modular indexing and LRU. However, as the number of sessions increase in web servers, these simple cache designs without considering web session characteristics cannot efficiently keep session data. A more efficient TCB cache is required to provide high cache performance.

2.1.2 Challenge in protocol processing on CPUs

In addition to TOEs, a large number of sessions also pose a challenge when the TCP/IP protocol stack is running on CPUs [14]. We establish a server-client environment, where the client opens the specific number of TCP sessions and sends 1KB requests across all of the sessions in a round-robin way to the server. Both the server and client are Intel machines with 2.67 GHz Intel Quad-core processors. Intel performance counters are used to instrument Linux in-kernel network stack and measure the execution time of individual kernel functions or groups of kernel functions. The lives of processing a request with one session and 4K sessions are shown in Figure 2a and 2b, respectively with a timeline scale of 500 CPU cycles per unit. The horizontal dashed line separates the kernel and user space, and only kernel functions are considered. Note that the figures only show functions in the TCP critical path and do not consist of functions in the non-critical path such as buffer allocation, de-allocation and scheduling etc.

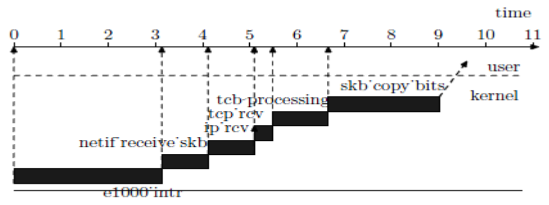

Figure 2a. Life of packet (single session)

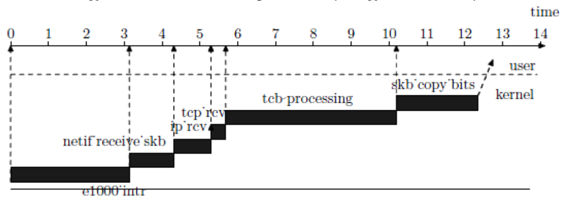

Figure 2b. Life of packet (4K sessions)

The received request processing starts from the interrupt handler *e1000_intr* in the device driver. After the interrupt handler, the request is delivered up to the IP layer (*ip_rcv*) and the TCP layer (*tcp_rcv*). Then, the network stack performs TCB lookups to find the destination TCB's address and does per-session processing according to TCB data, both of which we refer to as *TCB processing* in figures. Finally, the request is copied to user applications by using the *skb_copy_bits* function. The timing analysis shows that the *TCB processing* overhead increases rapidly with a large number of sessions, and becomes significant along with other two overheads in the TCP critical path: the driver and data copy. Since existing research [3, 10, 21] effectively reduces those two overheads, it becomes important to address the remaining *TCB processing* challenge. Our analysis shows that TCB lookups and access mainly contribute to the overhead of *TCB processing*. Web servers with a large number of sessions increase the chance that TCB data is polluted in caches, and degrade TCB lookup performance as well because traversing the linked list in a bucket is prone to incurring cache misses [14].

2.2 Characterization of Web Sessions

In the web domain, a web session is defined as a sequence of requests made by a single client during its visit to a particular server [2, 7]. A modern web page includes reference-indexed embedded files which are typically images or graphs; these files are required to properly display the web page to the client. Thus, a typical request for a web page usually results in multiple consecutive client requests for those embedded items. Extensive studies on real web traffics have shown that web sessions exhibit the *ON/OFF* model [2, 7]. The entire transfer period for the whole page is referred as *ON* period, and the time gap between two requests for two embedded items as Idle when server responses are transmitted. After the client receives the whole web page, it usually takes a period of time for the client to read the page before sending the next page request. This period is referred as the *OFF* period. During the *ON* period, TCB is frequently accessed, but no access occurs in the *OFF* period. Thus, keeping or not replacing cached contents during the *ON* period is critical, a property that is used later to design the *speculative* cache replacement policy.

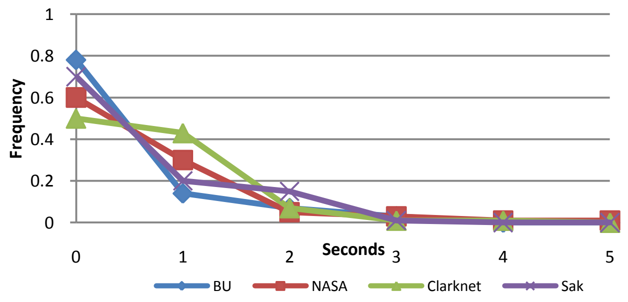

Figure 3a. Inter-request time frequency in *ON*

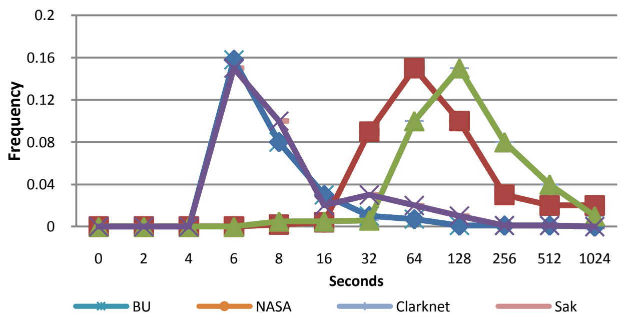

Figure 3b. *OFF* time frequency (*OFF*)

We choose four web server traces to study the characteristics of web sessions: Boston University trace (BU), NASA-HTTP (NASA), ClarkNet-HTTP (Clarknet), Saskatchewan-HTTP (Sak).

We measure both the time between two consecutive requests during the page transfer (in *ON*) and the time between two consecutive *ON* (*OFF* time) for all traces. Figures 3a and 3b show the frequency for the time. We observe that the inter-request time in the *ON* period is fairly small compared to the *OFF* time and is typically less than 1 second. The above time analysis guides us to design an efficient cache replacement policy.

2.3 Cache Indexing

Numerous papers [13, 26, 27, 31] have demonstrated that using alternative cache hashing functions for CPU caches can reduce cache conflict misses for SPEC CPU benchmarks by achieving a more uniform cache access distribution. We evaluated the performance of different hash functions such as traditional Modulo hashing (Mod), XOR-based hashing (XOR) [26, 27, 31], Prime Modulo (PMod) [13], Prime Displacement (PDisp) [13] and CRC [23]. We also studied multiple hashing schemes (each cache bank has a separate hash function), such as two Prime Displacement Hashing (2-Pdisp) [13], two XOR-based hashing (2-XOR) [26, 27], four Prime Displacement hashing (4-Pdisp), four XOR hashing (4-XOR) [26, 27]. For multiple hashing schemes, we use the cache replacement policy ENRU (Enhanced Not Recently Used) as used in [13]. The results are shown in the next section where we present the new TCB cache organization based on the *Universal* hash function and show its superiority over all others for the four web server traces.

3. A NEW TCB CACHE

In this section, we elaborate TCB cache designs considering web session characteristics. The cache organization is described in Subsection 3.1 and the bit selection is explained in Subsection 3.2. In Subsection 3.3, we illustrate the Lifetime array used by the new cache replacement policy, which is presented in Subsection 3.4.

3.1 Cache Organization

A cache organization is primarily defined depending on how a set is indexed. Our aim is to distribute the mapping uniformly that can ensure simultaneous occupancy of a large number of sessions being connected to the web server at a time. *Universal* hash functions are known to generate an even distribution of workload over the hash buckets and are relatively easy for hardware implementation [25]. We present the TCB cache miss ratios of four web server traces with various hash functions in Figure 4, where all cache miss ratios are normalized to the miss ratio of modulo mapping (Mod). We observe the following: 1) both Mod and XOR are not good fit for TCB cache; 2) PMod and PDisp are not as good as *Universal* and CRC; 3) having two hash functions obtains better performance than single hash function. It was observed in [13] that PMod and PDisp hash functions are better than Mod and XOR for SPEC CPU benchmarks. As we can see, they are also better for web server traces, but not as good as the proposed *Universal* hash functions. Among all of the hashing schemes, *2-Universal* achieves the best performance. It may be noted that having more than two hash functions degrades performance because more cache banks split the original LRU set and sacrifice the effectiveness of the cache replacement policy.

In order to understand the performance gap of various hash functions, we study probability distribution function (PDF) of absolute deviation of the number of sessions in cache sets (or |X minus expected value of X|, where X is the number of sessions in a cache set) and show result for one trace (Sak) in Figure 5. The figure points out that multiple hash functions have higher probability at small values like 50 and thus achieve a more even cache access distribution. Although other traces studies are not shown here, they behave similarly.

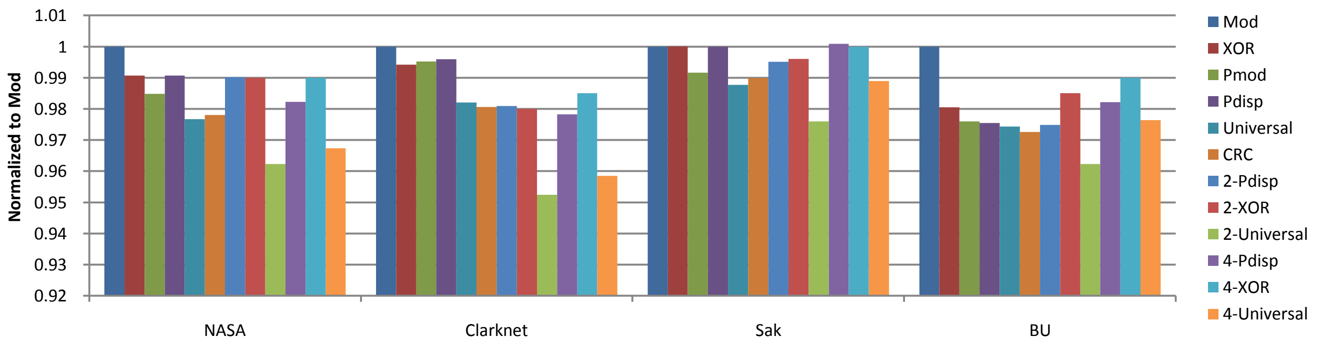

Figure 4. Performance of cache hash functions

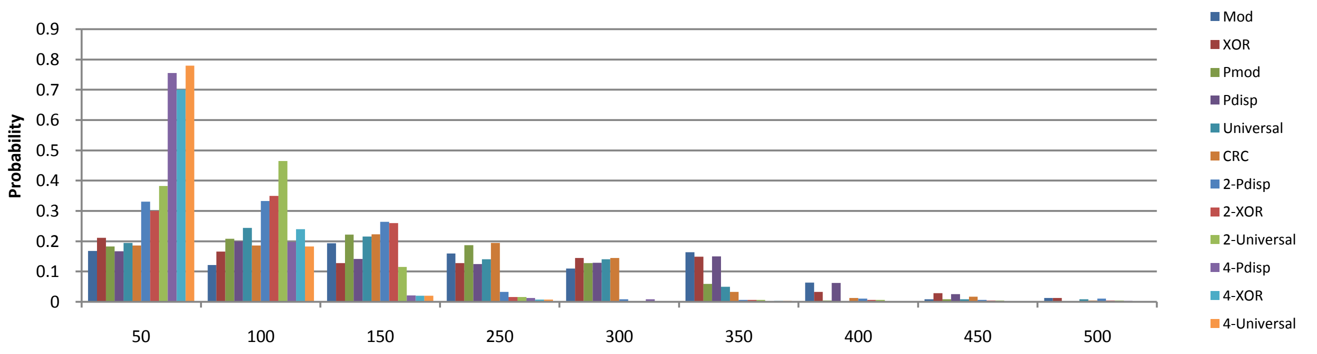

Figure 5. PDF of absolute deviation of #sessions in cache set

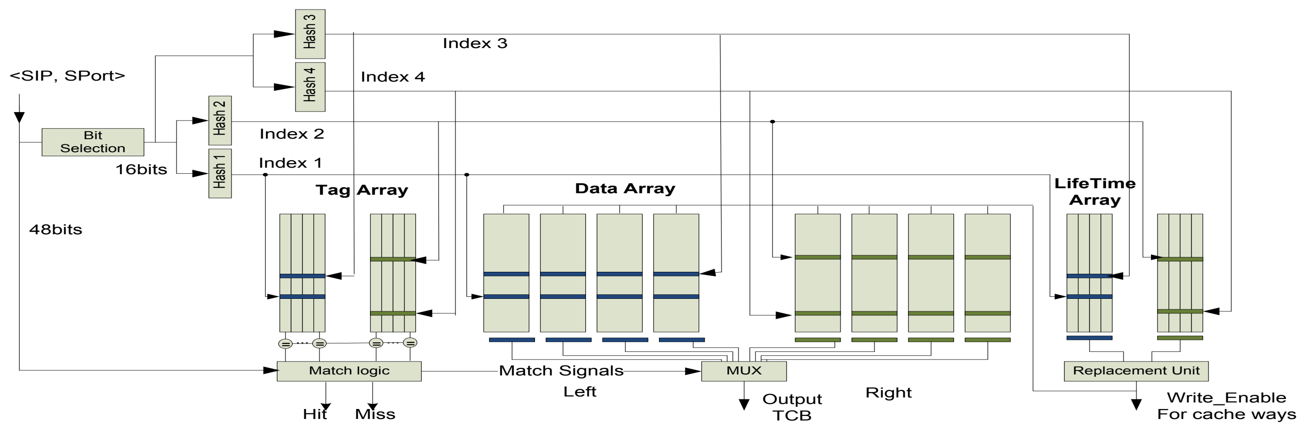

Figure 6. TCB Cache Architecture

Figure 6 illustrates the hardware design of the TCB cache, which is addressed by session identifiers using *Universal* hash functions. The TCB cache has tag arrays and data arrays as traditional CPU caches, but it adds a new Lifetime array to track the cache line's *ON/OFF* status, which is used by the hardware replacement unit. As observed in Fig. 4, two *Universal* hash functions (hash1 and hash2) being employed by two cache banks give the best miss ratio. Hence, we use two cache banks in Fig. 6, each consisting of a 4-way set associative cache. We also add two auxiliary *Universal* hash functions (hash3 and hash4) to be used by our cache replacement policy to migrate *ON* cache lines (see subsection 3.4). We do a bit-by-bit analysis of session identifiers and select 16 important bits as indexing bits to reduce *Universal* hashing hardware complexity. The selection process of the particular bits is described in the subsection 3.2. In order to access a session state, CPUs extract a 2-tuple from a packet header and issue an operation to the cache. The cache first locates the two cache sets corresponding to the two hashes (hash1 and hash2) of the 16 bits and then does the tag check with the 2-tuple in parallel. If the operation is hit in the cache, the session state is operated; otherwise, the cache uses auxiliary functions hash3 and hash4 to lookup the cache again. If not found, the hardware replacement unit is triggered to select a cache line for the new data. Since only a portion of a 2-tuple is used for hashing, the tag in each cache line is a full-fledged 2-tuple. We also include 4 bytes TCB memory address in tag arrays to make the TCB cache interact with the memory. Although TCB is a 512 bytes data structure, only a portion of data in each TCB is frequently accessed during processing packets [4, 14, 30, 32]. We use the full system simulator Simics [19] to study the frequency of access in Linux to TCB data and find that only ~64 bytes are frequently accessed. This is because most of the packets belong to the TCP fast path, requiring much fewer than the entire TCB data of 512 bytes. The similar observation have been made in TOEs that storing 64 bytes information for each session is sufficient to implement the offloaded processing tasks [9]. Therefore, we use a cache line of 64 bytes to keep those states.

3.2 Indexing Bit Selection

The two *Universal* hash functions in the TCB cache are from a function class called H_3, which has amenable hardware implementation [25]. Each hash function in H_3 is a linear transformation $B^T = QA^T$ that maps a w-bit binary string $A = a_1 a_2 \ldots a_w$ to an r-bit binary string $B = b_1 b_2 \ldots b_r$.

$$\begin{bmatrix} b_0 \\ b_1 \\ \cdots \\ b_{r-1} \end{bmatrix} = \begin{bmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,w-1} \\ q_{1,0} & q_{1,1} & \cdots & q_{1,w-1} \\ \cdots & \cdots & \cdots & \cdots \\ q_{r-1,0} & q_{r-1,1} & \cdots & q_{r-1,w-1} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \cdots \\ a_{w-1} \end{bmatrix}$$

Each bit of B is calculated as: $b_i = (a_1 \circ q_{i1}) \oplus (a_2 \circ q_{i2}) \ldots . (a_w \circ q_{iw}) \quad i = 1,2,\ldots,r,$ where $\circ$ denotes AND, and $\oplus$ denotes XOR circuits, respectively. In the TCB cache, w means the bits of a hash input and r is the bits of the cache index. Since hash functions in H_3 are the same except the parameter Q, each hash function can be configured from a generic chip by providing different parameters.

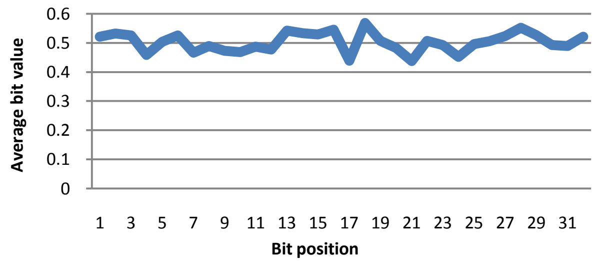

Figure 7a. Average bit value of IP address

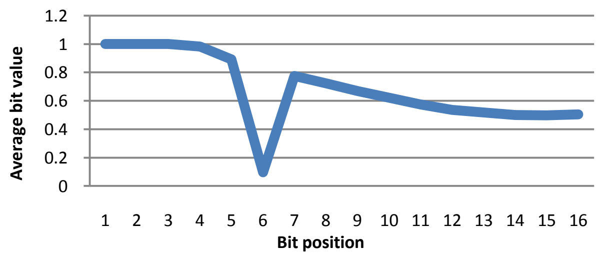

Figure 7b. Average bit value of port

Hashing latency and hardware complexity increase rapidly with increase in the input bits. We study bit distribution of session identifiers of web traces with the goal to reduce the number of input bits. We measure the average values of the bits distributed in IP address and port number and show them in Figure 7a and 7b (the first bit is the MSB). The best indexing bits (or important bits) should be those with an average value of 0.5; meaning that they are set 50% of the time over a large series of session identifiers. We notice that bits in IP address have similar importance but 8 least significant bits in port number are more important than other bits. That is mainly because ports start from 1024 (ports <1024 are assigned for system services) and are typically allocated within a limited range of 256, but IP address is distributed more randomly. Given these observations, we choose 8 bits from port and 8 bits from IP address as indexing bits, as shown in Figure 8. Experimental results in Section 4 show that tailored index bits can achieve the same performance as 48 bits 2-tuple.

Circuit implementation of calculating an output bit is illustrated in Figure 9 and each bit calculation is performed in parallel. The implementation needs 5 gate delays at most (1 gate delay in AND circuits and 4 gate delays in XOR circuits). Each gate only takes ~10 picoseconds with Intel 60nm fabrication technology [12] and thus 5 gate delays can be easily implemented within a single CPU cycle (1000 picoseconds per cycle for 1GHz CPU).

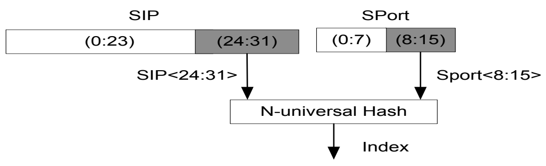

Figure 8. Bit selection

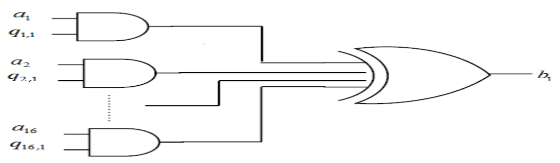

Figure 9. Circuit implementation

3.3 Lifetime Array

The Lifetime array is used to track the cache line's *ON/OFF* status and its structure is shown in Figure 10. In the Lifetime array, we maintain one 3-bit life counter for each TCB cache line to track the *ON/OFF* status. The most significant bit (MSB) of each 3-bit counter indicates *ON* or *OFF*. When MSB equals to 1 (111 to 100), it means *ON*, and 0 (011 to 000) means *OFF*. The counter is always initialized to the max value "111", and counted down every 1/4 second. After 1 second, the status switches to *OFF*, as the counter becomes "011". We choose 1 second as the threshold because according to the observation in Section 2.2, it is highly likely that web sessions are in *OFF* if they have not been touched for 1 second. The system countdown signal is triggered by a clock divider which basically counts the clock cycles and asserts a '1' by every N cycles. For example, let the system clock frequency (FREQ) be 2GHz and the *ON* period (T) 1 second. In order to get an 8Hz output, the N would be FREQ*T/4 = 500M cycles.

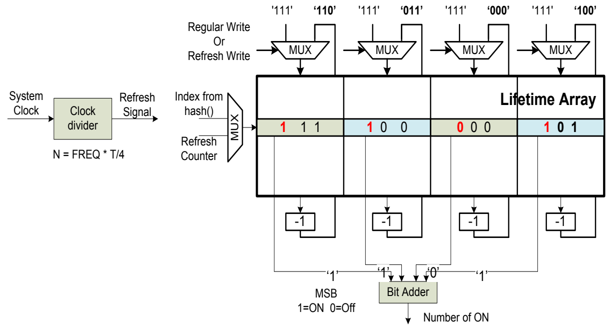

Figure 10. Lifetime array structure

There are two kinds of operations for the Lifetime array:

Regular read/write cycle: it happens at every TCB write. The corresponding life counter will be initialized to "111". Due to the possibility of cache replacement, we need to read out the original *ON/OFF* bits (MSBs of each counter) before the write. As in regular caches, we perform a read access in the first half cycle, and a write in the second half cycle. The read will collect the four *ON/OFF* bits, and sum them up through a bit-adder. The total number of *ON* will be sent to the hardware replacement unit.

Refresh write cycle: Similar to a DRAM memory refresh, which prevents the leakage of DRAM cells, we also perform a whole array scan once every 1/4 second. The difference is that, after reading the current value, we do not write the same value back, instead, it is reduced by 1 and is then written back. The only exception is "000", but 000-1=111, and thus we retain the value when the counter is zero. The refresh performance or power overhead is negligible, as hundreds of cycle vs 500 million cycles.

3.4 Speculative Cache Replacement Policy

Although multiple cache banks can reduce conflict misses, they make it difficult to implement cache replacement policies like LRU at reasonable cost and force using pseudo-LRU s [13, 26, 27, 31]. Topham *et al.* [31] presented a way to implement an affordable LRU for multiple cache banks by adding a timestamp to each cache line. Every time a cache line is accessed its timestamp is updated with the access sequence. When a miss occurs, the line with the least timestamp is replaced. They showed that an 8-bit timestamp achieves comparable performance for SPEC95 floating point benchmarks. However, we find that more than 24 bits are needed in the TCB cache in order to achieve good performance. What is more, more cache banks split LRU sets and sacrifice the effectiveness of LRU.

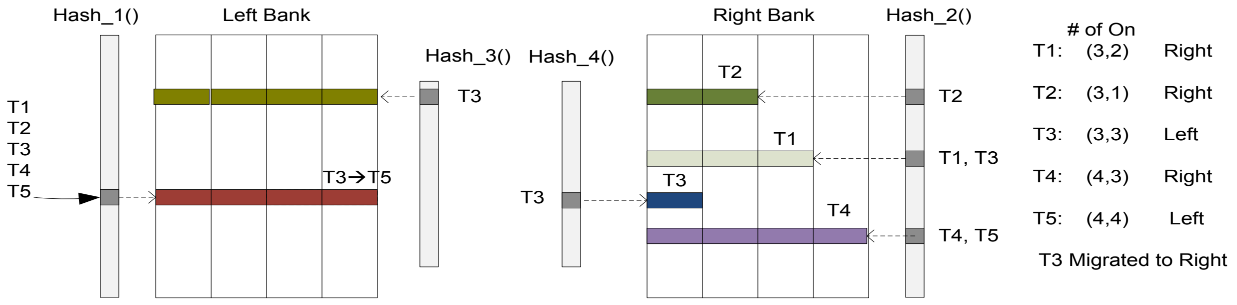

Figure 11. ***Speculative*** **replacement policy**

We design a *speculative* cache replacement policy by harnessing the *ON/OFF* model to address the above issues. Since a web session in the *ON* mode will be accessed very frequently, the policy aims to keep *ON* cache lines as long as possible as follows. 1) when a cache miss occurs, the policy selects a cache bank with fewer *ON* cache lines in two corresponding cache sets indexed by hash1 and hash2, in case of a tie, we choose the left cache bank for simplicity. It load balances *ON* cache lines among cache banks and increases the occupancy ratio of *ON* cache lines in the cache. We notice from in-depth studies that LRU is unaware of *ON* cache lines and may result in imbalance of *ON* cache lines among cache banks, and thus incurs unnecessary eviction of *ON* cache lines. 2) Inside each cache bank, if an *OFF* line is in the LRU position, we replace it for new data, otherwise, we check *ON* cache lines to find a migratable cache line (an *ON* cache line is referred to as migratable if there are *OFF* cache lines in its corresponding cache sets). A migratable cache line is randomly chosen and migrated to its corresponding cache set to keep *ON* cache lines in the cache as long as possible. The proposed scheme has some similarity with the hash-rehash scheme proposed long time back for direct-mapped cache, but our scheme uses different hash functions, multiple banks, migrates only selected replaced data. To increase the chance that we can find a migratable cache line, we introduce two auxiliary Universal hash functions (hash3 and hash4) to index the replaced *ON* cache line and migrate it to an *OFF* cache line if found. If an *OFF* cache line is not found during the auxiliary hash, the replaced cache line is discarded. Like lookup case, auxiliary hash hash3 and hash4 are simultaneously carried out for replacement. While sequential auxiliary hashing (or pipeline hashing) restricts cache access by hash1 and hash2, we notice that most of cache hits occur in the first hashing (hash1 and hash2) and the penalty is more than overcome due to increased cache hits. The sensitivity and performance impacts of our new policy are described later in Section 4.

Figure 11 illustrates one example of the *speculative* cache replacement policy. Suppose there are some *ON* TCBs in the TCB cache, which are colored but unlabeled. Given an access sequence of TCBs T1, T2, T3, T4, T5, the policy places T1, T2, T4 in the right cache bank and T3 in the left cache bank. When T5 comes, neither of two corresponding cachet sets in two cache banks has *OFF* cache lines and T3 is replaced. Since T3 is still in the *ON* mode, our policy gives T3 one more chance to stay in the cache by using two auxiliary hash functions, therefore T3 is migrated to the right bank for future accesses.

4. PERFORMANCE EVALUATION

4.1 Evaluation Methodology

We developed a trace-driven cache simulator to evaluate TCB cache designs. Four web server traces: Boston University trace (BU), NASA-HTTP (NASA), ClarkNet-HTTP (Clarknet), Saskatchewan-HTTP (Sak) are chosen for experiments because they are frequently used in network and architecture research. These traces contain all HTTP requests to the corresponding web servers during data collection periods.

In experiments, we denote the TCB cache in TOEs employing both LRU and modular hash as TCB (Mod). Since implementing LRU with two hash functions is complex, we evaluate a pseudo-LRU cache replacement policy ENRU for multiple cache banks similar to [13, 31]. We refer to the TCB cache with the pseudo-LRU and *2-Universal* as TCB (2-hash). Finally, we evaluate the proposed TCB cache with *2-Universal* and the *speculative* cache replacement policy and denote it as TCB (spec). Since our cache also implements a migration policy, we include the TCB cache without the migration scheme to understand the migration benefits and denote it as TCB(no-migrate). We test 1000 different *Universal* hash functions by randomly generating 1000 parameters and observe that they have similar performance within a range of 2.5%. We select the best hash parameters.

In addition, we study the performance benefits of using the new TCB cache in TOEs or integrating the cache into CPUs. We calculate the TCB access overhead (per packet miss ratio * memory latency) and incorporate it into the protocol processing time in [9] to study the performance impacts of the new TCB cache on TOEs. Furthermore, we use the full system simulator Simics by enhancing it with the detailed cache, I/O timing models and modeling of the effects of network DMA to understand the benefits of integrating the TCB cache into CPUs. Note that the integrated cache sits in parallel with L2 cache. Two networked systems (client and server) running Linux 2.6.16 are simulated. In the client, the replay tool opens multiple sessions to the apache server to simulate multiple clients and then generates requests from the web traces while keeping the same behavior inside each session. Since accesses to heap data structures among *tcp_v4_rcv* and *tcp_rcv_established* functions are for TCB items [4], we refer to those accesses as TCB accesses. We replace cache misses due to TCB accesses with cache misses of our TCB cache from our trace-driven cache simulator to approximate the performance benefits of integrating the TCB cache into CPUs. All caches in experiments have the same cache line size of 64 bytes with detailed simulator parameters listed in Table 1.

Table 1. System Parameters

Processor	Two cores, 3GHz, in-order, single-issue
ICache/DCache	32 KB 2-way, 2-cycle hit latency
L2 Cache	4M, 8-way split, 10 cycles hit latency
Memory	300 cycles
I/O register	800 cycles
TCB Cache	32KB, 10 cycles hit latency
NIC	LRO, 64 packets/interrupt

4.2 TCB Cache Performance

We study the performance of various cache configurations for all traces by comparing their cache miss ratios in Figure 12. We use TCB (Mod) as a baseline to understand the benefits of our optimizations. We observe that the baseline TCB (Mod) has a 56% miss ratio per packet with the BU trace. TCB (2-hash) reduces the miss ratio to 37% by achieving a more uniform cache access distribution. TCB (no-migrate) obtains a 32% miss ratio by load-balancing *ON* TCBs among cache banks. With speculative cache replacement policy, TCB (spec) achieves a smaller miss ratio of 28%, corresponding to 50% reduction compared to the baseline. Other three traces exhibit similar behavior. The NASA trace has a 50% miss ratio when it is run on the baseline system. Miss ratios are lowered to 33%, 28% and 26% when we run the trace on TCB(2-hash), TCB(no-migrate) and TCB (spec). Similarly, cache miss ratios for the Sak trace are 69% TCB (Mod), 55% TCB (2-hash) and 51% TCB(no-migrate).

TCB (spec) obtains a smaller miss ratio of 44%, corresponding to 37% relative reduction compared to TCB(Mod). When we come to the Clarknet trace, the miss ratios are 42% for TCB (Mod), 31% for TCB (2-hash) and 25% for TCB (no-migrate). TCB (spec) further reduces the miss ratio to 22% and achieves 47% cache miss reduction compared to the baseline. All above results verify the effectiveness of our cache indexing scheme and the *speculative* replacement policy.

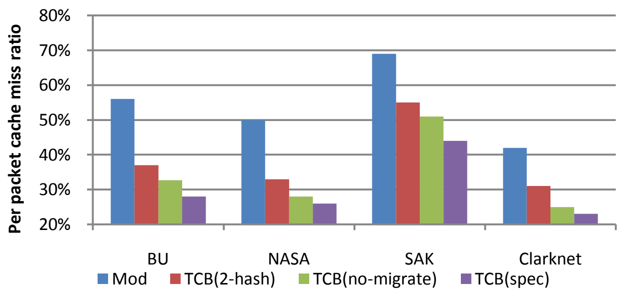

Figure 12. Per packet cache miss ratio

4.3 Impact of Bit Selection

To reduce the hardware complexity of *Universal* hash, 16 representative bits (IP<24-31> and Port<8-15>) are chosen for the TCB cache, as shown in Figure 8. In this subsection, we study TCB cache performance and justify the design of our 16-bit hash. We compare 16-bit hash with full-fledged 48-bit hash and other possible bit lengths hash. Since Port<0-7> is not as important as other bits of 2-tuple, we only consider all other 40 bits (organized as in Figure 8) for possible bit lengths. We present the cache miss ratio comparison in Figure 13, where n-bit represents a hash with the input of n least significant bits of the 40 bits and all miss ratios are normalized to the miss ratio of 48-bit hash. The figure shows that 8-bit hash degrades the performance but 16-bit hash is able to achieve the same cache performance as 48-bit hash while requiring the least hardware complexity. 16-bit hash lowers the hardware complexity, which allows the *Universal* hash to be feasibly deployed on on-chip caches requiring low hash latency and low power consumption. Circuit implementation shows that one output bit calculation in 48-bit *Universal* hash needs one 48-bit XOR logic and 48 AND logics, corresponding to 7 gate delays and 95 CMOS gates (47 gates in the XOR logic and 48 gates for AND logics). However, 16-bit *Universal* hash only uses one 16-bit XOR logic and 16 AND logics for calculating one output bit, corresponding to 5 gate delays and 31 CMOS gates (15 gates in the XOR logic and 16 gates for AND logics).

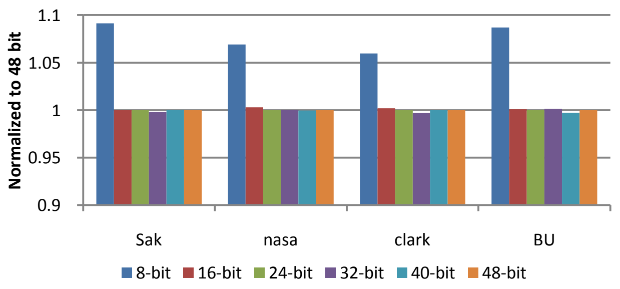

Figure 13. TCB performance of *n-bit* hash

4.4 Exploration of Cache Design Space

We also explore cache design space along three axes: cache replacement policies, cache size, set-associativity. We include three alternative replacement policies and denote them as TCB (RR), TCB (16), TCB (Access). TCB (RR) is the policy which chooses a cache bank for the new data in a round robin way. TCB (16) is the implementation of LRU with a 16-bit timestamp in each cache line. TCB (Access) selects the cache bank with less cache access to the two corresponding cache sets when a miss occurs. In Figure 14, all miss ratios are normalized to the miss ratio of the *speculative* replacement policy. We observe that TCB (16) has the similar miss ratios to TCB (RR) and TCB (Access) while it needs higher storage, and TCB(spec) achieves the lowest miss ratios for all four traces and only needs three extra bits for each cache line.

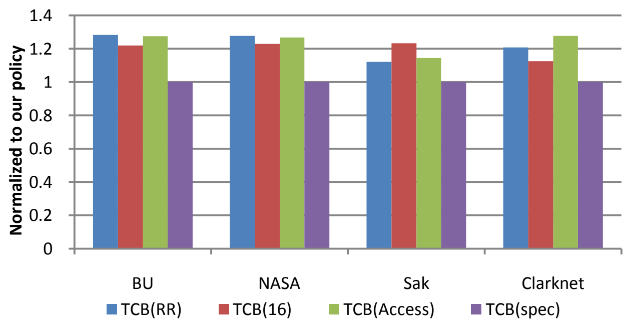

Figure 14. Performance impact of replacement policies

In addition, we present the TCB (spec) miss ratios over various cache sizes normalized over a 32KB cache, as shown in Figure 15. The figure shows that both 32KB and 64KB TCB cache sizes achieve good cache performance. When the cache size is reduced to 16KB and 8KB, the cache performance is dramatically degraded because of capacity misses. This study points out that 32KB is a suitable TCB cache size for web servers with thousands of concurrent sessions. We also evaluate the performance impacts of set-associativity of each cache bank on TCB(spec) as shown in Figure 16. We observe that both 4-way and 8-way achieve good cache performance over all four traces.

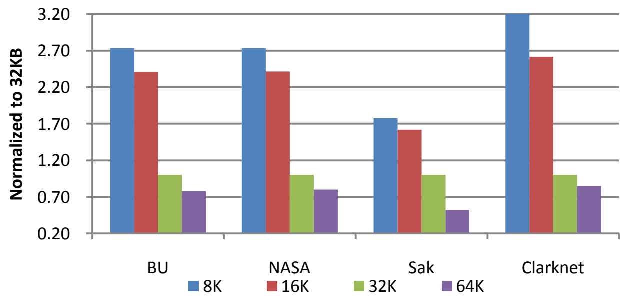

Figure 15. Performance impact of cache size

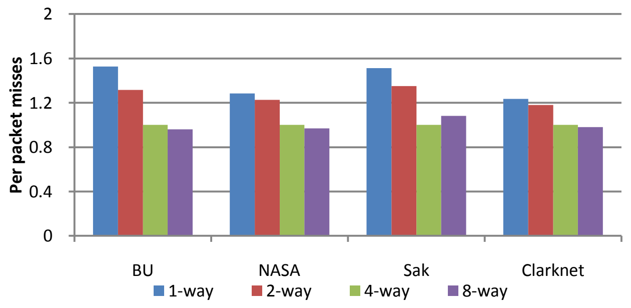

Figure 16. Performance impact of set-associativity

4.5 Using the New TCB cache

Our research resolves the issue of per-session data and is supplementary to existing approaches. First, the TCB cache can be applied to TOEs to replace the traditional TCB cache. Second, with the support of the TCB cache, DCA or Integrated NIC architectures are able to address the per-session data access challenge while running TCP/IP on CPUs.

We show the performance impacts of using the new TCB cache in TOEs on packet processing time in Figure 17. The results are normalized to the original TOE using the simple TCB cache. Our result projects that the new cache can reduce TCP/IP processing time by more than 20%. The reduced processing time will save web server response time. In addition, we also evaluate the performance benefits of integrating the TCB cache into CPUs in Figure 18 and 19. We use the optimization DCA delivering packets into L2 cache as the baseline configuration and denote it as orig. We normalize results to the baseline system. In the original system, frequently accessed TCB items are distributed across multiple cache lines and hence several cache misses could occur for one packet. Also, traversing linked lists due to TCB lookups is prone to incurring cache misses, deteriorating cache performance. By providing high cache hit ratios and avoiding linked list traversal with cache hits, the new TCB cache reduces TCP/IP request processing time by up to 23% and saves up to 5% web server response time.

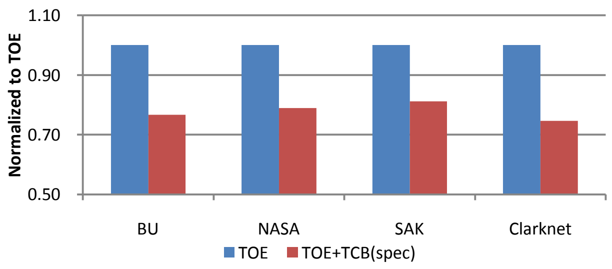

Figure 17. TCP/IP receiving time in TOEs

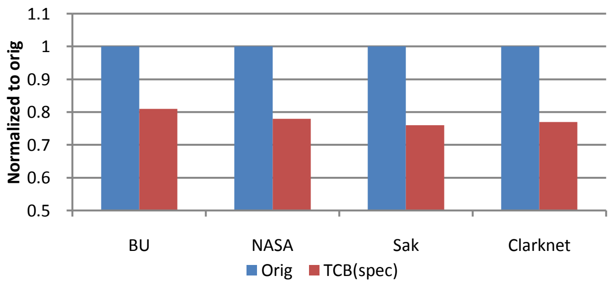

Figure 18. TCP/IP receiving time

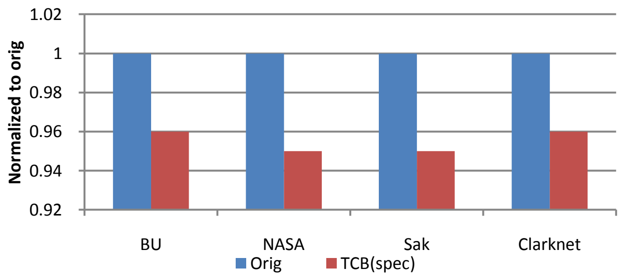

Figure 19. Web server response time

4.6 Discussion of Using the New TCB Cache

Since TOEs already use a dedicated TCB cache to accelerate accessing TCB, it is straightforward for TOEs to leverage a new TCB cache. Without extra hardware support, the new cache can be easily adopted to replace the traditional cache. In contrast to TOEs designed for network processing, integrating the new TCB cache into general purpose CPUs running the TCP/IP stack requires extensive architecture and system supports: 1) ISA needs to be extended to include cache Read/Write/flush instructions; 2) OS needs to use these new hardware instructions to access and manage the new cache, thus incurring troublesome instrumentation work. Hence, as of now, we believe that the new TCB cache is more suited for TOEs customized for network processing. However, when integrating NICs into CPUs like [3, 18] becomes popular in future, it is feasible for NICs or CPUs to leverage the new TCB cache.

5. RELATED WORK

5.1 Architectural Support for TCP/IP

It is well documented that Internet servers spend a significant portion of time processing packets [14, 18, 20, 32]. A wide spectrum of research has been done from the architectural perspective to resolve the overhead issue [3, 6, 9, 10, 15, 16, 21, 22, 28, 29, 32]. The essence of these studies has aimed at reducing the communication cost of CPUs and NICs. TOEs [6, 9, 22, 29] accelerate the protocol processing in NICs and improve performance by freeing up CPU cycles and reducing PCI traffics. Kim *et al.* [15, 16] offloaded some connections to TOEs for balancing CPUs and NICs. In contrast to TOEs, Binkert *et al.* [3] integrated a simplified NIC to reduce the communication cost by implementing zero-copy and reducing access latency to NIC registers. Intel proposed DCA to route network data into processor caches to reduce the packet access overhead [10, 11, 17].

While all of the aforementioned approaches can improve the processing performance, they ignored the per-session data. Typically, TOEs put a dedicated cache to manage per-session TCB data for providing fast access. However, the TCB cache is a traditional cache without any optimization. It is insufficient to manage a large number of web sessions and becomes a major bottleneck for packet processing. In addition, a large number of sessions also increase the per-session data access overhead while running TCP/IP on CPUs. Kim *et al.* [14] first showed that a large number of web sessions dramatically degrade TCP/IP performance because the working set size of session data structures grows in proportion to the number of sessions, simply increasing the L2 cache size would have limited benefits.

5.2 Cache Designs

There have been a large volume of studies done on CPU caches to reduce conflict misses by using alternative cache indexing functions [13, 26, 27, 31]. Seznec [26, 27] designed a skewed two-way set-associative CPU cache, where two different XOR-based hash functions are used for indexing the distinct cache bank, and showed its performance superiority over modular hash. By envisioning the benefits of XOR-based hash, Topham *et al.* [31] evaluated the performance of XOR-based hash for a number of different cache organizations and concluded that XOR-based hash is a promising indexing scheme to most cache organizations. Kharbutli *et al.* [13] studied the pathological behavior of various hash functions and applied two prime-based hash functions to L2 caches. Our paper extensively studies the performance of various hash functions and employs multiple *Universal* hash functions as TCB cache indexing. Result shows that *Universal* hash functions are more promising than any existing hash functions used in CPU caches. In order to deploy *Universal* hash on caches, we carefully study the bit distribution of session identifiers and tailor index keys. To couple with the new cache indexing, we design a *speculative* cache replacement policy by harnessing the *ON/OFF* model. Although the migration scheme is similar to the hash-rehash scheme proposed for direct-mapped caches [1], it employs *Universal* hash for rehashing cache lines and only migrates *ON* cache lines to *OFF* cache lines, avoiding eviction of valuable data.

6. Conclusion

In this paper, we conducted detailed TCP/IP studies from the per-session perspective and proposed a new TCB cache to efficiently manage per-session TCB data in web servers. The dedicated cache is designed to be addressed by a specified subset of session identifiers. To provide high TCB cache performance, we extensively study performance of various hash functions and employ a new *Universal* hash based cache indexing scheme with two independent cache banks. Some important bits are carefully selected as hash keys to reduce hashing hardware complexity. To further enhance the performance, we harness the *ON/OFF* model of web sessions to design a *speculative* cache replacement policy and employ migrating the replaced *ON* blocks to *OFF* region of the cache. Simulation results show that the new TCB cache can efficiently manages per-session data. By envisioning the benefits, using the new TCB cache in TOEs or even integrating it into CPUs can significantly reduce TCP receiving time and web server response time.

7. ACKNOWLEDGMENTS

The research was supported by NSF grants CCF-0811834, NEDG-0832108, CSR-0912850, and a grant from Intel Corporation. We would like to thank Prof Scott Rixner from Rice University for shepherding the paper. We also wish to thank the anonymous referees for their valuable feedback on this paper.

8. REFERENCES

[1] A. Agarwal, J. Hennessy, M. Horowitz, Cache Performance of Operating Systems and Multiprogramming, *ACM Transactions on Computer Systems*, Nov. 1998.

[2] P. Barford, M. Crovella, Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Measurement and Modeling of Computer Systems*, 1998.

[3] N. L. Binkert, A. G. Saidi, S. K. Reinhardt, Integrated Network Interfaces for High-Bandwidth TCP/IP. *ASPLOS* 2006.

[4] D. P. Bovet, M. Cesati, Understanding the Linux Kernel, Third Edition, O' Reilly Media.

[5] J. Carter, M. Wegman, Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 1979.

[6] Chelsio Communications. http://www.chelsio.com/.

[7] K. Claffy, Internet Workload Characterization. Ph.D. thesis, UC San Diego, June 1994.

[8] C. A. Cunha, A. Bestavros, M. E. Crovella, Characteristics of WWW Client-based Traces. Boston University Department of Computer Science, Technical Report TR-95-010, 1995.

[9] Y. Hoskote, B. A. Bloechel, G. E. Dermer et al., A TCP Offload Accelerator for 10Gb/s Ethernet in 90-nm CMOS, *IEEE Journal of Solid-State Circuits*, Vol 38. No.11, 2003.

[10] R. Huggahalli, R. Iyer, S. Tetrick, Direct Cache Access for High Bandwidth Network I/O. *ISCA*, 2005.

[11] A. Kumar, R. Huggahalli, Impact of Cache Coherence Protocols on the Processing of Network Traffic. *MICRO*, 2007.

[12] Intel Technology Journal, 130nm Logic Technology Featuring 60nm Transistors. Low-K Dielectrics and Cu Interconnects.

[13] M. Kharbutli, K. Irwin, Y.Solihin, J. Lee, Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses, *HPCA* 2004.

[14] H. Kim, S. Rixner, Performance Characterization of the FreeBSD Network Stack. CS Technical Report TR05-450, Rice University, 2005.

[15] H. Kim, S. Rixner, TCP Offload through Connection Handoff, *Eurosys*, 2006.

[16] H. Kim, S. Rixner, Connection Handoff Policies for TCP Offload Network Interfaces, *OSDI*, 2006.

[17] A. Kumar, R. Huggahalli, S. Makineni, Characterization of Direct Cache Access on Multi-core Systems and 10GbE. *HPCA*, 2009.

[18] G. Liao, L. Bhuyan, Performance Measurement of an Integrated NIC Architecture with 10GbE. *HotI* 09, USA.

[19] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgen, G. Hallberg et al., Simics: A Full System Simulation Platform. *IEEE Computer*, February 2002.

[20] S. Makineni, R. Iyer, Architectural Characterization of TCP/IP Packet Processing on the Pentium M Microprocessor. *HPCA*, 2004.

[21] S. S. Mukherjee, B. Falsafi, M. D. Hill, D. A. Wood. A Coherent Network Interfaces for Fine-Grain Communication. *ISCA* 1996.

[22] Q. Nhon T, P. Ramesh, F. Jean Marc, US Patent 7,406,087, Systems and Methods for Accelerating TCP/IP Data Stream Processing.

[23] W. W. Peterson, D.T. Brown, Cyclic Codes for Error Detection. In *Proceedings of the IRE*, January 1961.

[24] F. Pong, Fast and Robust TCP Session Lookup by Digest Hash. *ICPADS*, 2006.

[25] M. Ramakrishna, E. Fu, E. Bahcekapili, Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Trans on Computers*, 1997.

[26] A. Seznec. A Case for Two-way Skewed Associative Caches. *ISCA* 1993.

[27] A. Seznec. A New Case for Skewed-associativity. IRISA Technical Report #1114, 1997.

[28] L. Shalev, V. Makhervaks, Z. Machulsky et al., Loosely Coupled TCP Acceleration Architecture, *HOTI* 2006.

[29] C. C. Sharp, US Patent 7,287,092, Generating A hash for A TCP/IP Offload Device.

[30] R. Stevens, TCP/IP Illustrated Volume 1, Addison-Wesley Professional.

[31] N. Topham, A. Gonzalez, J. Gonzalez. Eliminating Cache Conflict Misses through XOR-based Placement Functions. *ISC* 1997.

[32] L. Zhao, R. Illikkal, S. Makineni et al., TCP/IP Cache Characterization in Commercial Server Workloads. *CAECW-7*, 2004.