DHash: A Cache-Friendly TCP Lookup Algorithm for Fast Network Processing

Kai Zhang[1], Junchang Wang[2], Bei Hua[3], Li Lu[4]
University of Science and Technology of China(USTC), Hefei, Anhui, 230027, China
Suzhou Institute for Advanced Study, USTC, Suzhou, Jiangsu, 215123, China
{[1]kay21s, [2]wangjc, [4]luly9527}@mail.ustc.edu.cn, [3]bhua@ustc.edu.cn

***Abstract*—A typical hash based TCP lookup algorithm is hard to make a trade-off between speed and space. This paper presents DHash, a high-efficient TCP lookup algorithm that aims at supporting large number of sessions in high speed networks. DHash achieves this goal by designing a compact and cache-friendly lookup data structure that well fits the modern computer architectures. To show the power of DHash, we implement it in a user-space TCP/IP stack, and then parallelize the stack on the Intel multicore processors. Experiments show that DHash is able to achieve 16.3Mpps while handling one million concurrent sessions on our parallel platform.**

***Index Terms*—TCP lookup; Cache-friendly Hash Table; High-performance Network Processing;**

I. Introduction

Developing PC-based network devices with off-the-shelf commodity hardware and general-purpose operating system is attractive due to low cost per performance and easy programmability. However, keeping them at high speed (e.g., 10Gbps) is challenging. Recently software routers built on commodity PCs have achieved noticeable performance [17] [10], whereas the same thing does not happen on layer 4+ devices such as servers, application-layer firewalls, intrusion detection systems, proxies, and load balancers, etc. These devices differ from routers in that they mainly work upon TCP layer to do stateful packet processing which imposes significant overhead on system.

TCB (Transmission Control Block) is a data structure used to maintain the state of each TCP session. In a typical TCP implementation, TCBs are organized in a hash table with conflicts resolved by linked lists. Generally, the size of a TCB is 280∼1300 bytes [7]. When there are a great number of concurrent sessions, e.g., one million sessions – a general metric for nowadays commodity high-end network devices [6] [1], TCBs will take up 280MB∼1.3GB memory space. Since the scale of the last level cache(LLC) is typically 10MB in mainstream commodity processors, the memory footprint of TCBs is tens of hundreds of times the size of LLC. With such a huge working set, almost each search step along a linked list will incur a LLC miss. Previous researches [8] [21] [26] have identified that packet processing time is dominated by CPU stalls on main memory access, other than the execution of instructions. Consequently, even minor memory accesses may increase the packet processing time significantly. In addition, in a 10Gbps network, the time budget for a minimum Ethernet packet (64 bytes) is only 67ns which roughly equals to the time of a DRAM access, therefore the bottleneck of TCB lookup needs to be removed straight away.

This paper presents DHash (Digest Hash), a cache-friendly TCB lookup algorithm that speeds up TCB lookup by decoupling it from TCB access. Instead of organizing TCBs in a hash table and identifying each TCB with its session identifier, i.e., the 4-tuple (source IP address, destination IP address, source port, destination port), DHash uses a short signature that is the digest of session identifier to mark a session, and only includes session signatures in the hash table. With only short signatures rather than full TCBs storing in the lookup table, table size is greatly reduced, and cache locality is improved. With this cache-friendly design, our algorithm can achieve high performance on modern computer architectures.

As one single core is incapable of handling a great number of sessions at very high speed, to verify DHash in a challenging environment, we implement DHash in an open source TCP/IP stack and parallelize the stack on multicore architectures (in the experiments, we use an eight-core server). By adopting a series of parallelizing and optimizing techniques, especially the connection-affinity principle that distributes all the packets of a session to the same core, the parallelized TCP/IP stack achieves scalable performance improvements on the platform. The parallelizing work shows that DHash is easy to parallelize and scales well with the core number, both of which are important features for migrating legacy network applications to multicore architectures.

This paper has the following two major contributions:

- A novel hash table data structure and algorithm called DHash is designed specifically for fast TCB

This paper is supported by the Fundamental Research Funds for the Central Universities under Grant No.WK0110000007.

lookup. It is equivalent to a hash table with very low load factor, but consumes much less memory space compared with traditional hash table.
- A parallelized TCP/IP stack enhanced with DHash is built on a multicore server. It can handle one million TCP sessions at 16.3Mpps, which is higher than 14.88Mpps - the maximum packet throughput on a 10Gbps link.

To the best of our knowledge, no literature has reported a TCB lookup algorithm achieving this speed with such a large scale.

Road map of the paper is as follows. Section II summarizes our experimental study on traditional TCB lookup algorithm and surveys related work. Section III elaborates the design and implementation of DHash, whose theoretical analysis is given in Section IV. Section V gives the single core performance of DHash. Section VI describes a parallel run-time system reports DHash's parallel performance. Section VII concludes this paper.

II. Background and Related Work

Network equipments are known to spend a significant portion of time on protocol processing. Performance characterization of TCP/IP stack has been studied intensively [21] [18] [5]. This section analyzes existing TCP lookup algorithms and outlines the related works.

A. Bottleneck of Traditional TCP Implementation

Hash table is the most widely used method to organize TCBs in current TCP implementations. When a TCP packet arrives, its TCP session identifier (i.e., the 4-tuple) is used to generate a hash value, which in turn is used to locate a hash bucket, and then used to search along a conflict linked list. Hash table is a good candidate for fast lookup only when the load factor is low. In the case of millions of concurrent TCP sessions, the size of TCB hash table would be exceptionally large if the load factor is kept low; however, restricting hash table size may greatly increase hash collisions, which counteract the benefit of hash table.

TABLE I
Ratio of Processing Time Taken by Different Processing Functions

Number of Session	I/O Cost	IP Proc.	TCP (w/o TCB)	TCB Proc.
10	4.1%	4.6%	32.5%	58.8%
131,072	0.14%	0.16%	1.1%	98.6%

We set up a testing environment with a TCP/IP implementation from Libnids 1.24 (with 1040 buckets as its default implementation) [2], and an optimized NIC driver from IOEngine [17]. We measure the processing time of the constituent modules with 10 and 131,072 sessions, respectively. The functions include packet I/O, IP processing, TCP processing excluding TCB lookup, and TCB lookup.

Experimental results are shown in Table I, where TCB lookup is the most significant bottleneck in the bottom four layers of the stack, which takes 58.8% of the total processing time with 10 sessions, and 98.6% with 131,072 sessions. The experimental data also denotes that TCB lookup time is sensitive to the number of TCP sessions, indicating that the traditional TCB lookup algorithm lacks scalability.

B. Related Work

Previous studies [21] [25] [26] reveal that system performance deteriorates rapidly with enlarged number of TCP sessions, because TCB working set grows in proportion to the number of sessions, resulting in poor spatial locality. Furthermore, as TCB access lacks temporal locality when a large number of sessions are active, simply increasing the cache size has limited benefits [21].

To improve the worst-case performance of hash table lookup, some optimized implementations [19] [16] [13] are proposed to reduce both access time and memory usage. However, none of them was designed from the very beginning to fit the features of modern computer architectures, especially the effective use of cache which dominates the application performance in real systems. HashCache [4] implements a fixed-size, non-chained hash table in disk that maps the cached object name to the location of the corresponding file on disk. Different from a normal hash table, new object is simply written over a previous object if the table overflows.

Many hardware solutions are proposed to speed up TCB lookup. Liao et al. [12] design a server-dedicated hardware TCB cache by utilizing the characteristics of web sessions. Fong [11] designs a complex function that transforms the session signature and TCB location into a 32-bit code which are stored in a specialized hardware TCB cache. TOE [20] [22] [23] is another kind of hardware solution that offloads TCP processing into NIC to free up CPU cycles and reduce PCI traffic. Due to limited resource, all of the above solutions are designed to process only a small number of sessions (less than 100K) and is very expensive to expand. In addition to the difficult trade-off between performance and price, hardware solutions require modifying existing network stack implementations, which makes the technology difficult to deploy.

III. Design Space Exploration

Our goal is to design an efficient TCB lookup algorithm that is capable of handling one million TCP sessions

in 10Gbps networks, and a cache-friendly TCB lookup algorithm called DHash is designed.

A. Data Structure

The linearly growing TCB working set, which stems from tight coupling of TCB lookup and TCB access via a shared hash table, is the root of low performance and poor scalability of existing TCB lookup algorithm. Based on this observation, our ***first*** design point is to decouple TCB lookup from TCB access by designing a novel lookup data structure, and making it as compact as possible.

Linked list usually has low spatial locality, especially in a system with insufficient last-level cache. Therefore, our ***second*** design point is to replace the linked list with continuous storage space in resolving conflicts. We divide the hash bucket into fixed-sized slots to store the matching keywords. To make good use of cache, hash buckets are set to be the size of a cache line (64 bytes in mainstream commodity processors), and meanwhile each of them is initialized to be cache line aligned.

To make the lookup table compact, our ***third*** design point is to compress the information as much as possible. A 32-bit signature rather than the 96-bit session identifier is used as the matching keyword of a session. Furthermore, TCB locations are not explicitly stored in the lookup table, but calculated from the location of their corresponding slots. In DHash, A TCB array is pre-allocated and mapped as follows: the jth slot in the ith hash bucket is mapped to a TCB whose index is $i \times ElemPerBucket + j$ in the array.

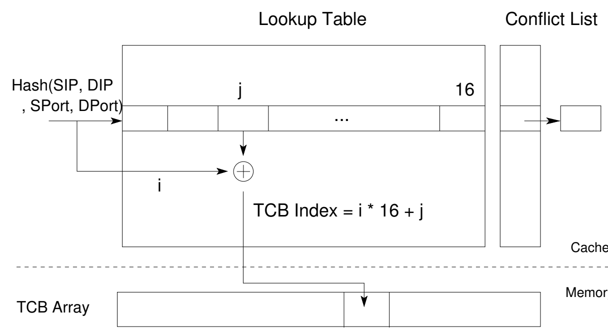

Fig. 1. Data Structure of DHash

Figure 1 sketches DHash's TCB lookup data structure for processors with 64-byte cache line size. TCP session identifier is used as the input of the hash function. Each hash bucket contains 16 slots, and each slot is 32 bits long. The TCB array is pre-allocated, where the first N elements (N equals to the number of slots in the hash table) have static one-to-one mapping to the slots in lookup table, and the rest ones are reserved as a TCB pool for future allocation. When the number of conflicted TCBs is larger than the maximum number of slots in a hash bucket, extra TCBs are allocated from the TCB pool, and their locations are explicitly stored along with their signatures in the corresponding conflict list.

B. Signature Algorithm

Using shorter signatures reduces hash table size and speeds up lookup, but it may also introduce false positive, i.e., different session identifiers happen to have the same signature. Therefore, whenever a session signature is matched in the lookup table, the corresponding TCB is accessed and the 4-tuple is compared to confirm an actual match. As false positive causes extra memory accesses, low false positive rate is the most important characteristic of a signature algorithm. In addition, because signature algorithm must be performed on each TCP packet, low computational overhead is the second important requirement.

We aim at designing fast and simple signature algorithms, and here are four example signature algorithms with popular fast hash operations: XOR and CRC32 (x86 processors have incorporated CRC32 instruction in the SSE4.2 instruction set). 1) The first algorithm simply performs XOR on the 4-tuple to get the signature, i.e., $(SrcIP \oplus DestIP \oplus SrcPort \oplus DestPort)$. 2) The second algorithm firstly performs $(SrcIP \oplus DestIP)$ and $(SrcPort \oplus DestPort)$ separately, and then concatenates them to calculate the CRC value. 3) The third algorithm inverts the order of CRC and XOR operation. It firstly performs $CRC(SrcIP|DestIP|SrcPort \oplus DestPort)$ and $CRC(DestIP|SrcIP|SrcPort \oplus DestPort)$, and then performs XOR to get the signature. All the three algorithms are symmetric. 4) The fourth algorithm solely uses CRC. It firstly performs $CRC(SrcIP|DestIP|SrcPort|DestPort)$ and $CRC(DestIP|SrcIP|DestPort|SrcPort)$ separately, then concatenates the most significant 16 bits from the two CRC values to form a 32-bit signature.

As zero is used to mark an empty slot, a specific 32 bits numeric value is assigned as the signature if the calculated result is zero. For asymmetric signature algorithms, when the calculated signature doesn't match for the first time, we swap the signature's upper and lower 16 bits and do the comparison again. We will measure the false positive rate of each signature algorithm in Section VI-C, and give the theoretical analysis on DHash's false positive rate in Section IV.

C. Reduce Comparison Times with Major Location

When a hash bucket is hit, signatures in it are compared one by one to find a match. If the comparison always begins with the first slot, the average number of comparisons is eight, i.e., half of the slot number in a bucket. To reduce the number of comparisons, we adopt

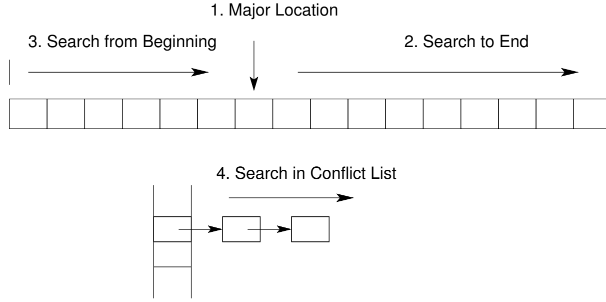

Fig. 2. Procedure of Finding a Session

a mechanism similar to Major Location [24] to arrange for the storage of signatures in the hash bucket. The basic notion is that each TCB signature has a preferred slot to store, and the lookup process begins with that slot hoping to locate it quickly.

$$(Sig \ \& \ 0x0F) \tag{1}$$

$$(Sig \ \& \ 0x0F) \oplus ((Sig \ \& \ 0xF0000) >> 16) \tag{2}$$

When a session identifier is hashed to a bucket, its major location is calculated and served as the starting point of the search. With 16 slots, 4 bits are needed to indicate a location. To make the two directions of a TCP session search from the same starting point, major location should be symmetric. There are several ways to obtain symmetric major location. For symmetric signature algorithms, any 4 bits of the signature can be extracted to form the major location. Function (1) is an example that takes the last 4 bits of a symmetric signature to form the major location. For asymmetric signature algorithms (e.g., Algorithm 4), function (2) gives a possible method that takes bits 0-3 and bits 16-19 of the signature and XOR them to form the major location.

D. Detailed Algorithm Description

1) Find a Session: Figure 2 illustrates the procedure of finding a session. When a TCP packet is received, its session identifier is used to calculate the hash index and session signature, and then major location is calculated from the session signature (Fig.2-1). Signature comparison begins with the slot indicated by major location and moves towards the end of the bucket (Fig.2-2); it wraps back at the end of the bucket and restarts from the first slot (Fig.2-3) if necessary. When a signature is matched, the corresponding TCB is accessed, and the complete 4-tuple is further compared. If this is a false positive, search procedure continues. If no match is found in the hash bucket (Fig.2-4), the corresponding conflict list (if any) is checked. The procedure returns with an index of TCB or *NOT FOUND*.

2) Add a Session: The procedure of adding a session, whose mission is to find an empty slot, is similar to that of finding a session. When an empty slot is found in the hash bucket, the new session signature is stored in it, and its corresponding TCB is returned. If no empty slot is found in the hash bucket, an empty TCB is obtained from the TCB pool, and a new element containing session signature and TCB location is linked to the conflict list.

3) Delete a Session: When a session is closed, its TCB must be released. If the signature is found in hash bucket, this session can be deleted by clearing the slot to zero. If the session signature is found in the conflict list, DHash first puts TCB back into the pool, and then removes the list element.

IV. Theoretical Analysis of DHash

We start with the equivalent hash tables shown in Figure 3. *Table-A* is a two-level hash table formed by two hash functions, where the first level table has N buckets and each second level table has 2^b buckets. *Table-A* is equivalent to a one-level hash table with $N * 2^b$ buckets (*Table-B*).

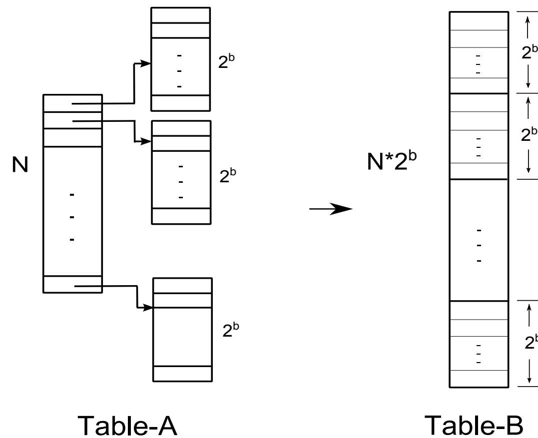

Fig. 3. Equivalent Data Structures

In the case of DHash, the signature algorithm can be considered as the second hash function that generates b bits as the hash index. However, the hash index is not used for locating a hash bucket, but is recorded to identify an object, which can be viewed as resolving conflicts with open addressing. Therefore, the false positive rate in each bucket in DHash equals to the conflict rate of the second level hash table in *Table-A*.

A. Load Factor

Performance of a hash table largely depends on its load factor. Because *Table-B* is an equivalent data struc-

ture to DHash's lookup table, the load factor of DHash equals to that of *Table-B*, where N is the number of bucket in the lookup table and b is 32.

If M TCP sessions have been maintained in the lookup table and the load is uniformly distributed, the load factor of the DHash's lookup table can be calculated as $(M/N) * (1/2^b)$. In DHash, $b = 32$ and M/N is no more than 16, so we get

$$(M/N) * (1/2^b) \leq 3.72 * 10^{-9}$$

In other words, DHash is equivalent to a hash table with extremely low load factor.

DHash greatly reduces the memory consumption than the traditional hash table. For example, with one million elements, to obtain a load factor of $3.72 * 10^{-9}$, a traditional hash table needs $2,000$ TB for the bucket headers in a 64-bit system, while DHash is pre-allocated with only 4.5MB ($(64 + 8) * 62,500$ Bytes) in total.

B. False Positive Rate

As mentioned before, the false positive rate in each hash bucket in DHash equals to the conflict rate of the second level hash table in Table-A. Each second level hash table in Table-A has $n = 2^{32}$ buckets, and the average number of session signatures contained in the table is no more than 16. Assume k session signatures have been maintained in the second level hash table, and E_k is defined as the event that k sessions do not collide in the table, its probability is calculated as

$$\begin{aligned} Pr\{E_k\} &= 1 \cdot (\frac{n-1}{n})(\frac{n-2}{n}) \cdots (\frac{n-k+1}{n}) \\ &= 1 \cdot (1 - \frac{1}{n})(1 - \frac{2}{n}) \cdots (1 - \frac{k-1}{n}) \end{aligned}$$

With $n = 2^{32}$ and $k <= 16$, the items $-\frac{1}{n}, -\frac{2}{n}, ..., -\frac{k-1}{n}$ can be considered as infinitesimal, and the approximation of e^x by $1 + x$ is quite good: $e^x = 1 + x + \Theta(x^2)$ [9]. Therefore,

$$\begin{aligned} Pr\{E_k\} &\approx e^{-\frac{1}{n}} e^{-\frac{2}{n}} \cdots e^{-\frac{k-1}{n}} = e^{-\sum_{i=1}^{k-1} i/n} \\ &= e^{-k(k-1)/2n} \approx 1 - k(k-1)/2n \end{aligned}$$

According to inclusion-exclusion principle, the probability that at least two sessions conflict equals to 1 minus the probability that no sessions conflict. The expected false positive rate in each hash bucket in DHash is

$$1 - Pr\{E_k\} \approx k(k-1)/2n \leq 2.79 * 10^{-8}$$

To sum up, DHash achieves extremely low false positive rate and load factor with much lower memory consumption compared with traditional hash tables.

V. Single Core Performance of DHash

To evaluate the performance of DHash, we borrow the open source TCP/IP stack from Libnids, and substitute DHash for the original TCP lookup algrotihm. The sequential performance of the system is evaluated in this section, and a parallel version is built and evaluate in Section VI.

A. System Configuration

We build the runtime system on a server equipped with two Intel E5620 quad-core processors running at 2.4GHz. Each processor has an integrated memory controller that supports 1066MHz DDR3 memory and is installed with 4GB memory. Each processor has a 12MB L3 cache that is shared by all the cores within that processor. The two processors are connected via QuickPath Interconnect (QPI) at 5.86GT/s. The operating system is 64-bit Fedora 14 with a stock Linux kernel (version 2.6.39.2). To obtain the maximum processing speed of DHash, packets are read from memory instead of the real NIC. Both the runtime system and DHash are compiled by GCC 4.5.1 with -O2 option.

To reduce the access to extra conflict lists, the expected number of sessions hashed to each bucket in DHash should be no more than 16. To support one million TCP sessions and cope with some load imbalance, we set the hash table size to $100,000$ buckets, which makes the expected number of sessions in each bucket to be 10. Therefore, the lookup table is pre-allocated 6.4MB memory space. Totally 7.2MB memory space is pre-allocated for DHash, including 0.8MB memory space for TCB pool.

Based on the analysis in Section IV, with $M = 10^6$ TCP sessions and $N = 10^5$ hash buckets (Section V-A), the load factor of the lookup table is $M/(N * 2^b) = 10^6/(10^5 * 2^{32}) \approx 2.3 * 10^{-9}$. The expected false positive rate of each bucket ($k = M/N = 10$ and $n = 2^{32}$) is $1 - Pr\{E_k\} \approx k(k-1)/2n \approx 1.05 \times 10^{-8}$. And the expected false positive rate of DHash ($k = 10^6, n = 10^5 \times 2^{32}$) is $1 - Pr\{E_k\} \approx k(k-1)/2n \approx 1.16 \times 10^{-3}$.

B. Trace File

Evaluation of DHash needs very large packet trace files that contain millions of concurrent TCP sessions. However, such trace can only be collected from Internet backbone or very large data center, and is not publicly available due to privacy and business secret protection. Therefore, we synthesize packet trace files with real-world session identifiers obtained from our campus gateway and MIT Lincoln Lab [3] to form different number of concurrent sessions. To simulate the worst-case traffic pattern, the TCB reusing distances in the synthesized trace files are maximized to stress DHash, i.e., in a trace file with N TCP sessions, every two packets from the same session are separated by *N-1* packets from other sessions.

In the generated trace files, each session has 20 packets that cover the life cycle of a TCP session.

TABLE II
CHARACTERISTICS OF TRACE FILES

	Pkt. Num.	Pkt. Len.	Max. TCP Sessions	Avg. TCP Sessions
File-1	5,242,880	67B	262,144	249,036
File-2	10,485,760	67B	524,288	498,073
File-3	20,971,520	67B	1,048,576	996,189

Three trace files are generated with different sizes, whose characteristics are shown in Table II. Column *Pkt. Num.* is the total number of packets in the trace file, and *Pkt. Len.* is the average packet size. The last two columns are the maximum and average number of concurrent sessions at run time.

C. DHash Lookup Performance

This section evaluates DHash's lookup performance and compares it with that of the original algorithm with the three trace files. The average DHash lookup time (in nanoseconds) of DHash is measured and shown in Figure 4, where *Orig.(N Bucket - M Memory)* represents the original algorithm with *N* buckets which take *M* memory space for bucket headers.

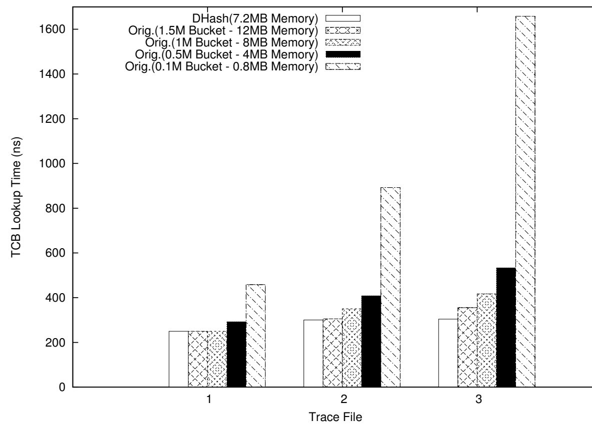

Fig. 4. TCB Loopup Performance

Experimental data shows that performance of original algorithm varies significantly with the TCP session number, whereas performance of DHash is very stable. This scale insensitive feature is very important for a working system to provide predictable performance for practical applications.

We noticed that the sequential implementation of DHash only achieves 3.3Mpps with File-3, probably due to insufficient power of one single core. To evaluate Dhash in a more challenging environment, we parallelize the TCP/IP stack with Dhash on a commodity multicore server [14] [15], and test the system with File-3 again in the next section.

VI. PARALLEL PERFORMANCE OF DHASH

A. Parallel Runtime System Setup

The parallel runtime system is built on the pipelined Run-To-Completion model shown in Figure 5. Each pipeline is divided into two stages named Input (IP) and Application (AP), which are assigned to different cores. Connection affinity is applied as an important principle in workload distributing, which guarantees that packets belonging to the same session are processed by the same AP core. Therefore, a symmetric hash function is used in the load balancing module. According to the system configuration, processing of a packet roughly includes three functional modules: (1) packet input and load balancing; (2) L3 processing including IP header validation and defragmentation; (3) TCP processing.

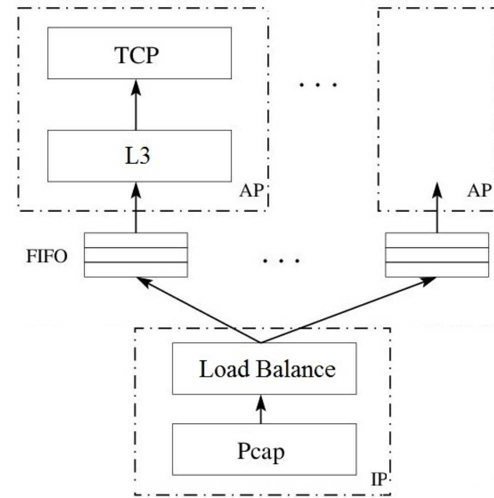

Fig. 5. The Pipelined RTC Model of the Parallel System

To decide on how many pipelines to use, the average execution time of each pipeline functional module is measured. In addition, as load balance (LB) that dispatches packets and FIFO that connects neighboring cores also sit on the data path, their costs are measured as well. The execution time shown in Table III is obtained when File-1 (see section V-B) is used as the input. Packets are loaded into memory in advance, and then fed into system for processing.

TABLE III
AVERAGE EXECUTION TIME OF KEY MODULES

(CPU Cycle)	Pcap	LB	FIFO	L3	TCP
Ex. Time	35	30	50	100	700

Let T_{IP}, T_{FIFO}, and T_{AP} represent the execution time of *IP* stage, enqueue/dequeue operation, and *AP*

stage, respectively. Using the data in Table III, T_{IP} and T_{AP} can be calculated as follows:

$$T_{IP} = T_{Pcap} + T_{LB} = 65(cycles)$$
$$T_{AP} = T_{L3} + T_{TCP} = 800(cycles)$$

Assuming that packets are evenly distributed among the AP cores, optimally seven pipelines are needed to make the IP stage and AP stage balance, since $(65+50 = 115) \approx ((800+50)/7 = 121)$. However, as there are only eight cores in our server and one core is reserved for OS, maximally six pipelines are used in our experiments.

B. Parallel Performance

For performance comparison, File-3, which contains one million concurrent TCP sessions, is used as input. Hash table size of DHash is set to be 100,000 buckets, as described in Section V-A. TCP implementation from Libnids is used as the traditional algorithm. As the default TCB lookup implementation in Libnids is sequential and only has 1040 buckets, for fair comparison, we parallelize it on our parallel framework and tune the hash table size to achieve maximum performance. For example, we increase the hash table size to 10,000, 100,000, 1,000,000 buckets, and so on. Figure 6 shows the performance of DHash and traditional TCB lookup algorithm with different hash table sizes. The horizontal axis represents the number of cores used, and the vertical axis represents the system performance in Mpps.

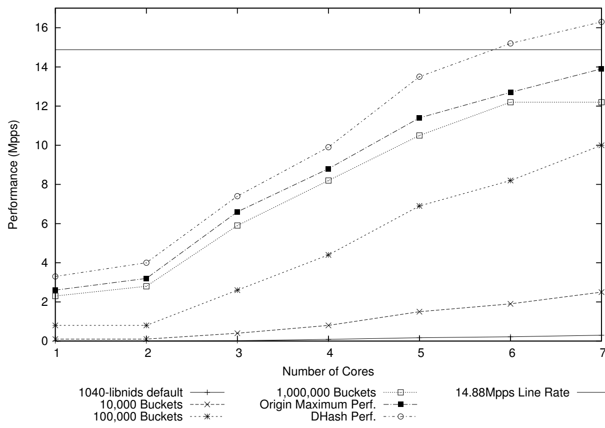

Fig. 6. System Performance with DHash and Original TCB Lookup Algorithm

In the case of traditional algorithm, system performance rises with the increase of hash table size until a threshold is reached (1.5 million buckets in our experiment). The reason is that the average length of conflict list gets shorter when larger hash table is used; however, when the hash table size exceeds the size of last level cache, cache thrashing happens. A maximum throughput of 13.90Mpps is achieved with seven cores and about 1.5 million hash buckets (about 12MB). Nevertheless, the throughput is still lower than 14.88Mpps, which is the maximum packet rate on a 10Gbps Ethernet link.

DHash achieves 15.2Mpps on six cores and 16.3Mpps on seven cores, and takes up 7.5MB memory space (including lookup table and conflict lists) in total. Of the one million sessions, only 11,413 TCB signatures (about 1%) enter the conflict lists, therefore accessing conflict lists can be considered as a rare case.

C. Impact of Signature Algorithm

This section measures the false positive rate of the four signature algorithms presented in Section III-B, and evaluates its impact on system performance.

TABLE IV
COMPARISON OF SIGNATURE ALGORITHMS

	Alg.1	Alg.2	Alg.3	Alg.4
Perf.(Mpps)	15.9	16.1	15.2	16.3
False Positive Rate(%)	0.18	0.10	0.33	0

Experimental data on the four algorithms listed in Table IV shows that higher false positive rate leads to lower system performance, which is sensible as false positive incurs extra memory access. All the four algorithms have very low false positive rate, which are on the same order of magnitude as the theoretical analysis result in Section IV.

It is worth noting that the experimental data in Table IV should be taken as a qualitative assessment, since it may bear some relationship to the trace files, and we do not intend to make a good-bad assessment of the four algorithms here.

D. Improvement Made by Major Location

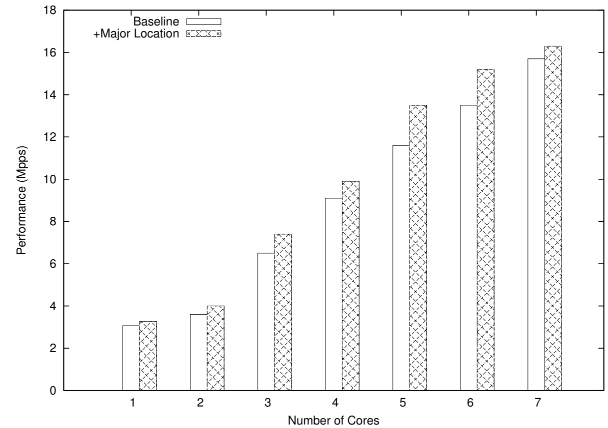

Fig. 7. Improvement Made by Major Location

Performance reported so far is obtained when the major location scheme is applied. This section evaluates the performance improvement made by major location. Figure 7 shows the system throughput achieved with and without major location scheme when different number of cores are used. It shows that on average 10% performance improvement is achieved with this scheme.

TABLE V
MAJOR LOCATION EVALUATION

	Perf. (Mpps)	First Hit Ratio	Average Comparison Times
w/o ML	15.7	-	6.8
w ML	16.3	67%	3.0

Table V compares the maximum performance (with six pipelines), first hit ratio (only major location scheme applies), and average comparison times of the two schemes. After major location is applied, the first hit ratio is as high as 67%, which reduces the average comparison times from 6.8 to 3.0 and leads to 0.6Mpps throughput improvement.

VII. CONCLUSION

This paper presents DHash, a software-based high performance TCB lookup algorithm that aims at handling large number of TCP sessions in high speed networks. Two optimization techniques are employed in the design and implementation of DHash. The first is to reduce the memory footprint of TCB lookup data structure by decoupling TCB lookup with TCB access, and replacing full TCB identifiers with shorter session signatures in lookup table. The second is to increase the memory access efficiency by replacing random access with sequential access when solving hash collisions. Experiments show that DHash is less sensitive to increased session number, and can achieve 16.3Mpps on our parallel platform.

REFERENCES

[1] "Cisco Catalyst 6500 Series Content Switching Module," "http://www.cisco.com/en/US/products/hw/modules/ps2706/products_data_sheet09186a00800887f3.html".

[2] "Libnids," "http://libnids.sourceforge.net/".

[3] "MIT Lincoln Lab," "http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html".

[4] A. Badam, K. Park, V. S. Pai, and L. L. Peterson, "Hashcache: Cache storage for the next billion," in *USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[5] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt, "Performance analysis of system overheads in tcp/ip workloads," in *Proc. 14th Ann. Intl Conf. on Parallel Architectures and Compilation Techniques*, 2005, pp. 218–228.

[6] Cisco, "Cisco Firewall Services Module for Cisco Catalyst 6500 and Cisco 7600 Series," "http://www.cisco.com/en/US/prod/collateral/modules/ps2706/ps4452/product_data_sheet0900aecd803e69c3.html".

[7] Cisco Corporation, "Defenses Against TCP SYN Flooding Attacks," 2010, "http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_9-4/syn_flooding_attacks.html".

[8] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of tcp processing overhead," *IEEE Communications Magazine*, vol. 27, pp. 23–29, 1989.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[10] M. Dobrescu, N. Egi, K. Argyraki, B. gon Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[11] Fong Pong, "Fast and robust tcp session lookup by digest hash," in *Proceedings of the 12rd IEEE International Conference on Parallel and Distributed Systems*, 2006.

[12] G. Liao, L. N. Bhuyan, W. Wu, H. Yu, and S. R. King, "A new tcb cache to efficiently manage tcp sessions for web servers," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2010.

[13] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, "Chisel: A storageefficient, collision-free hash-based network processing architecture," in *Proceedings of The 33rd Annual International Symposium on Computer Architecture (ISCA 33*. IEEE Computer Society, 2006, pp. 203–215.

[14] J. Wang, H. Cheng, B. Hua, and X. Tang, "Practice of parallelizing network applications on multi-core architectures," in *Proceedings of the 23rd International Conference on Supercomputing*, 2009, pp. 204–213.

[15] K. Zhang, J. Wang, B. Hua, and X. Tang, "Building high-performance application protocol parsers on multi-core architectures," in *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, 2011.

[16] S. Kumar, "Segmented hash: An efficient hash table implementation for high performance networking subsystems," in *Proceedings Symposium on Architecture for Networking and Communications Systems (ANCS'05*, 2005, pp. 91–103.

[17] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: A gpu-accelerated software router," in *Proceedings of the SIGCOMM'10 Conference*, 2010.

[18] S. Makineni and R. Iyer, "Architectural characterization of tcp/ip packet processing on the pentium m microprocessor," in *Proceedings of 10th International Symposium on High-Performance Computer Architecture*, 2004.

[19] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," in *Proceedings of the SIGCOMM'05 Conference*, 2005, pp. 181–192.

[20] H. Wilson, J. Xu, and N. Borkar, "A tcp offload accelerator for 10 gb/s ethernet in 90-nm cmos," in *IEEE Journal of Solid-State Circuits*, 2003.

[21] H. youb Kim and S. Rixner, "Performance characterization of the freebsd network stack," Rice University, Tech. Rep., 2005.

[22] H. youb Kim and S. Rixner, "Connection handoff policies for tcp offload network interfaces," in *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[23] H. youb Kim and S. Rixner, "Tcp offload through connection handoff," in *Proceedings of EuroSys Conference*, 2006.

[24] C. Zhang, X. Zhang, and Y. Yan, "Two fast and high-associativity cache schemes," in *Proc. 30th IEEE/ACM International Symposium on Microarchitecture*, 1997.

[25] L. Zhao, R. Illikkal, S. Makineni, and L. Bhuyan, "Tcp/ip cache characterization in commercial server workloads," in *Proc. Seventh Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2004.

[26] L. Zhao, S. Makineni, R. Illikkal, R. Iyer, and L. Bhuyan, "Efficient caching techniques for server network acceleration," in *2004 Advanced Networking and Communications Hardware Workshop*, 2004.