A Lock-Controlled Session Table Partitioning Scheme with Dynamic Resource Balancing for Multi-Core Architecture

Wen-Yen Tsai*, Nen-Fu Huang*+, Senior Member, IEEE, and Hsien-Wei Hung*

Institute of Communications Engineering*, Department of Computer Science+
National Tsing Hua University, Taiwan

Abstract — Connection tracking by manipulating session tables is essential for stateful inspection capable applications such as stateful firewalls, network-based intrusion prevention systems (NIPS), traffic accounting and monitoring to process packets according to session state information. With the prevalence of multi-core computing, it is crucial to optimize the existing connection tracking structures and algorithms to fully utilize the underlying parallelism. In this paper, we propose a lock-controlled session table partitioning scheme accompanied with a dynamic resource balancing algorithm for session-aware multi-core networking systems. Experimental results show that the proposed scheme reduces the number of lock contentions to a maximum of 100 times less and, in turn, boosts the performance to 3.5 Gbps higher than the baseline. 100% resource utilization is also achieved by overcoming the constraint of fixed-sized partitioning.

Keywords— Connection tracking; Stateful inspection; Multi-core Architecture

I. INTRODUCTION

Connection tracking is a fundamental part of any session-aware networking applications. By associating each packet to its corresponding connection, more intelligence can be put for packet inspection to even accelerate the processing speed based on connection-level information. Each time when a packet passing through, the connection tracker retrieves a $n$-tuple from values of source IP, destination IP and, for L4 protocol like TCP or UDP, the source port and destination port to calculate a digested value used to lookup up the session table. If no corresponding entry in the table is found, after some necessary integrity checks, the tracker allocates a new entry to insert it into the session table. Otherwise, the associated entry is consulted and updated. Hash table and search trie are most frequently used structures to implement session table for fast lookup. Moreover, appropriate pruning of stale or embryonic sessions and releasing their table entries is important to avoid the explosion of session table which severely degrades connection tracking efficiency and even opens a security hole for attacker

With the popularity of the ever-evolving modern multi-core platforms, networking systems should be designed in mind to achieve maximum aggregative performance by paralleling packet processing tasks. However, to avoid buggy race conditions and ensure correctness for applications running on the multi-core platform, synchronization mechanisms must be enforced. Locking and atomic operations are two frequently used techniques to achieve mutual exclusion while at the same time prevent the high-cost of context switches compared to sleeping semaphores. Unfortunately, synchronization alone is not the silver bullet and comes at a price. First, excessive synchronization operations lower the overall system performance due to their large overheads. Second, according to Amdahl’s law, multi-core system performance is maximized by minimizing serial code paths [1]. However, too coarse-grained locking and code path that is serial in intrinsic prevent parallelism. Third, even in the case that the application code can be parallelized in a much higher degree by fine-grained locks, program complexity grows exponentially and it introduces a great risk of system deadlock. Notwithstanding the lock order is guaranteed to be correct to avoid deadlock, overheads caused by lock contentions and cache line bouncing among CPU cores can greatly degrade the system performance.

In this paper, we propose a lock-controlled session table partitioning scheme which “localizes” connection tracking structures and their corresponding locks to each group of CPU cores in order to reduce the number of lock contentions and cache line bouncing. Accompanied with the dynamic resource balancing algorithm, resources are balanced according to their utilization rates in each group to achieve the same connection tracking capacity without session table partitioning. The entire connection tracking system is implemented in a Linux kernel module and its effectiveness is testified by the professional Ixia testing equipment.

The rest of this paper is organized as follows. In Section II, we briefly describe some related works of session table manipulation and multi-core packet processing. Section III elaborates the proposed partitioning scheme and resource balancing algorithm. Section IV illustrates the experimental results. Finally we conclude this paper in Section V.

This work was supported by the NSC projects under the grant numbers NSC 97-2221-E-007-108-MY3 and NSC 99-2219-E-007-007.
II. RELATED WORK

Several works for fast session table manipulations are proposed. Li, et al.[2] enhanced the PATRICIA trie employed in some session table implementations to improve the timeout (age-out) process. Zhang et al.[3] added a pointer pointing to the node of hash collision linked list matched successfully most recently to speed-up connection lookups by taking advantage of the IP flow locality. Two well-known open source packet filtering solutions, packet-filter (PF) [6] and Netfilter [7] integrated in the BSD* and Linux networking stack, respectively, base on the connection tracking technology to provide feature-rich and efficient packet filtering functionalities. Nevertheless, most of the works did not take advantage of the architectural parallelism to achieve superior performance on multi-core platforms.

On the other hand, the topic of multi-core packet processing is in active research. Fulp and Farley [5] introduced a new firewall architecture consisting of multiple firewalls configured in parallel that collectively enforce a security policy. OS vendors are also highly involved in developing their next-generation networking stacks to take fully advantage of the power of multi-core computing. The FreeBSD SMPng project [8] provides numerous resources about the implementation of the FreeBSD Symmetric Multi-Processing (SMP) OS kernel and its multithreaded networking stack. Receiver Side Scaling (RSS) of the scale networking initiative [9] proposed by Microsoft tries to join efforts from hardware NIC vendors with the Windows 7 and Vista networking stack to provide more scalable and high-performance packet processing capabilities on multi-core platforms.

III. SESSION TABLE PARTITIONING AND DYNAMIC RESOURCE BALANCING

A common way of efficient protocol implementation is to pre-allocate memory for the related data structures. This approach comes with the advantages of avoiding the dynamic allocation overheads and no need to define complex policies for allocating memory among subsystems during system runtime. In addition, deciding the maximum capacity of resource in advance, such as the maximum number of allowed TCP connections, eases the implementation and makes the behavior of the system predictable. Therefore, we adopt the pre-allocation strategy and partition connection tracking memory into several sub-tables and structures. Then, during system runtime, these resources are balanced among different groups of CPU cores according to resource utilization rates.

A. Session Table Partitioning

A typical connection tracking implementation on a multi-core platform should consist of the following structures:
1. A hash table or search trie for connection lookups
2. A pool of free connection entries
3. LRU lists for recycling expired entries
4. Accessory data structures such as statistics counters
5. Locks for protecting each table or structure

Given a global hash table with $N_{\text{hash}}$ buckets, and let $N_{\text{core}}$ be the total number of CPU cores in the system. The CPU cores are grouped in a fixed size, $N_{\text{race-core}}$, and each core in the same group contends for the same connection tracking structures and locks. The global hash table is thus partitioned into $N_{\text{core}} / N_{\text{race-core}}$ sub-tables and each has $N_{\text{hash}} / N_{\text{core}} * N_{\text{race-core}}$ buckets. The same method applies for partitioning the global free connection entry pool. In this manner, connection tracking is localized to each group of CPU cores to minimize the number of lock contentions and cache line bouncing.

Per-CPU Data and Avoidance of Cache Pollution

In order to implement the localization of the partitioned structures to each group of CPU cores, we use the Per-CPU data programming primitives introduced in Linux 2.6 kernel as the method for accessing local data of each CPU on large SMP machines. C pointers pointing to the addresses of per-CPU specific structures are stored in per-CPU data. By this technique, pointers of different CPU cores in a group can point to the same address to share the same structure. In addition, CPU brings L1 cache line-byte, e.g., 128 bytes, data into L1 data cache each time. Hence, when the size of data structures which two per-CPU pointers point to fall into one single L1 cache line, access to any of the structure by one CPU core will “pollute” the cache of the other core and cause it to invalidate the corresponding cache line. We align the data structures in L1 cache line size to avoid the cache pollution problem.

Port/Core Binding and Selection of $N_{\text{race-core}}$

By setting the SMP IRQ affinity of each network port, we can statically “pin” network ports to CPU cores. All ingress and egress packets from that port will be processed by the “pinned” CPU core. This kind of binding is fixed once the SMP IRQ affinity is decided and inflexible. IRQ balancing mechanism, as shown in Fig. 1(c), eliminates this inflexibility by redistributing the traffic to the busy and idle CPU cores. However, it introduces tremendous cache line bouncing due to frequent switches of packets belonging to the same connection among CPU cores. We adopt a method to bind each port to different CPU core and wraps around to the already bound cores if the number of ports exceeds that of CPU cores. A daemon is developed to monitor link status and packet Rx/Tx statistics and, accordingly, reassign SMP IRQ affinity of each port if necessary. Because the daemon binds only active ports to CPU cores, it avoids the problem of completely static binding. In addition, since active ports are assigned to the same cores each time in a high probability, cache misses due to packet switches are minimized.

To select a proper value of $N_{\text{race-core}}$, the simplest approach is for all the cores in the system to contend for the same global structures, i.e., $N_{\text{core}} = N_{\text{race-core}}$, as illustrated in Fig. 1 (a). Because this method causes the largest number of lock contentions and cache line bouncing, we just treat it as the baseline for performance comparison. Oppositely, as Fig. 1(d) shows, a front-end dispatcher can distribute packets belonging to the same connection to the same CPU core. In this extreme case, each core can have its own connection tracking structures (i.e., $N_{\text{race-core}} = 1$) without any lock. The demerit is that one or
more CPU cores must be used as the dispatcher and thus reduce the number of usable CPU cores. Problems of unbalanced dispatches and traffic burstness also degrade the performance. We propose to select $\text{N}_{\text{race-core}}$ according to the port configuration of the real application. Take IPS and VirusWall for example, most of these devices in the market are designed to operate in pairs of port, or segments. One port of each segment is connected to LAN and the other to WAN. In this configuration, packets of the same connection should appear in one of the two ports alternatively. Therefore, we can pin ports of one segment to two adjacent CPU cores in a physical CPU package for them to share the same connection tracking structures, i.e., $\text{N}_{\text{race-core}} = 2$, as illustrated in Fig. 1(b). This approach achieves maximum performance since on most architectures CPU cores in the same physical package share the last level cache (LLC) and incurs no cross-CPU data access penalties. Similarly, for devices that support link aggregation, packets belong to a connection will appear among the ports that form a trunk. For example, if a trunk is composed of four ports, it is appropriate to set $\text{N}_{\text{race-core}} = 4$ and pin the ports and cores accordingly.

**B. Dynamic Resource Balancing**

To overcome the problem of unbalanced resource utilization due to fixed sized partitioning, we propose an algorithm to dynamically migrate connection tracking resources.

**Resource Migration Operations and Policy**

Free connection entries can be linked in a list and migrated by first determining an appropriate number of entries to move and then splicing a sub-list from the lightly-loaded group to that of the heavily-loaded one as illustrated in Fig. 2. Resource balancing between two groups can thus be accomplished by a single list splice operation, or if there are other kinds of resources to balance, they can be easily moved in a similar manner. We don’t consider balancing hash table entries, because other connection tracking implementations use structures like search tries to perform connection lookup. In these cases, only connection entries need to be balanced.

Concerning resource balancing policy, several factors must be considered. First, appropriate timing to perform resource balancing should be determined. If the time interval is chosen to be too short, large overheads may be introduced due to excessive resource migration operations. On the opposite, an overlong time interval may cause connection tracking to be inefficient or even fail due to unbalanced resource distribution. We choose to perform resource balancing at the time points when connection aging-out are performed. For it introduces better timeliness than waiting a number of connections to terminate until, for example, TCP FIN or RST packets are received. Moreover, previous researches [4] proposed criteria of selecting age-out time intervals. Second, resource utilization is quantified by defining a utilization rate for each resource. A threshold $T$ corresponding to each rate is also defined to suppress unnecessary resource balances. We use a simple formula to calculate the utilization rate as the ratio of the number of used resources to the total ones. Last, we have to choose an appropriate number of resources to migrate. To transfer resources from group $j$ to $i$, let the utilization rate of $j$ and $i$ be $\rho_{j}$ and $\rho_{i}$, and $\text{N}_{j}$ be the number of free resources in $j$. Naturally $\rho_{i} > T$ and $\rho_{j} > \rho_{i}$ and therefore the number of resources to migrate is calculated as: $\text{N}_{i} * (\rho_{i} - \rho_{j})$. In the extreme case of $\rho_{i} - \rho_{j} = 1$, all free resources are migrated. Algorithm 1 summarizes the resource balancing algorithm.

**Resource Balancing Overhead**

Let $\text{N}_{\text{free-conn}}$ be the size of free connection entries of group $j$, the cost of finding the sub-list head to splice is $O(\text{N}_{\text{free-conn}})$, and then it takes $O(1)$ to splice the list starting from it. Other operations such as terminating the free connection entry list at the position of the sub-list head can be accomplished in $O(1)$ time. Therefore, given $\text{N}_{\text{core-group}}$ groups of CPU cores in the system, the total complexity of migrating free connection entries is as follows.

$$\sum_{j=0}^{N_{\text{core-group}}-1} \sum_{j=0}^{N_{\text{free-conn}}-1} \text{O}(\text{N}_{\text{free-conn}}) * \Delta_{p - T_{\text{free-conn}}} * \Delta_{\rho - \rho_{i}}$$

where $\Delta(a) = 0$, for all $a \leq 0$, and 1 for all $a > 0$. 

**Fig.1.** (a) 1 to 1 port/core binding with $N_{\text{race-core}} = 4$; (b) 1 to 1 port/core binding with $N_{\text{race-core}} = 2$; (c) IRQ balancing with $N_{\text{race-core}} = 4$; (d) Packet dispatcher with $N_{\text{race-core}} = 1$
Free conn. entry list of group j

Free conn. entry list of group i

Fig. 2. Splice a sub-list from CPU core group j to i.

Algorithm I. The resource balancing algorithm

Method: balanceConnEntry
1. for each group j of CPU cores
2. \( N_{\text{used,conn}} \) \( \leftarrow \) # used connection entries of i
3. \( N_{\text{total,conn}} \) \( \leftarrow \) # total connection entries of i
4. \( L' \) \( \leftarrow \) free connection entry list of i
5. \( \rho' \) \( \leftarrow \) \( N_{\text{used,conn}} / N_{\text{total,conn}} \)
6. if \( \rho' > T_{\text{free,conn}} \) then // utilization rate > balancing threshold
7. for each CPU core group j \( \neq i \)
8. \( \rho \) \( \leftarrow \) \( N_{\text{used,conn}} / N_{\text{total,conn}} \)
9. if \( \rho > \rho' \) then
10. acquire locks of CPU core group owned by i and j
11. migrate_cnt \( \leftarrow (N_{\text{total,conn}} - N_{\text{used,conn}}) \times (\rho' - \rho) \)
12. \( L'' \) \( \leftarrow \) sub-list containing migrate_cnt list entries in \( L' \)
13. terminate \( L'' \) at the head of \( L' \) and splice \( L'' \) to \( L' \)
14. \( N_{\text{total,conn}} \) \( \leftarrow \) \( N_{\text{total,conn}} - \text{migrate_cnt} \)
15. \( N_{\text{total,conn}} \) \( \leftarrow \) \( N_{\text{total,conn}} + \text{migrate_cnt} \)
16. release CPU core group locks owned by j and i

IV. EXPERIMENTAL RESULTS

In this section, we conduct several experiments to show the effectiveness of the proposed session table partitioning scheme in terms of reducing lock contents. Then the resource balancing algorithm is testified by a biased traffic profile.

The experiment hardware platform is based on a Portwell NAR 7100 board equipped with two Intel Xeon E5540 quad-core CPUs (8MB L3 cache) and 8GB DDR III memory. Eight Intel PCIe gigabit NICs supporting PCI MSI-X are used for network connectivity. The software installation is a Linux distribution with 2.6.29 SMP kernel capable of configuring SMP IRQ affinity of each NIC. We wrote a Linux kernel module to implement the entire connection tracking system for efficiency and future integration into the Netfilter framework. The kernel module is hooked in a very early stage of the kernel networking stack and treats the eight NICs as four segments for receiving and transmitting packets. It deals with basic packet integrity check, protocol de-multiplexing, connection table lookup, insertion and aging-out. After a packet is associated to a connection, operations like updating connection timestamp and/or tracking TCP state transition are performed before forwarding the packet. Hash table with collision chaining is used for session lookup, and UDP and TCP individually consumes a hash space of 32,768 buckets. Numbers of UDP and TCP session entries are set to 550,000 and 220,000, respectively. We adopt a coarse-grain locking approach to protect the connection tracking structures by a single spinlock contented by each CPU core in the same group, since fine-grain locking should exhibit the same results for our scheme. As suggested in [4], the smallest TCP connection aging-out interval should not be less than one second, so we setup a one second kernel timer to trigger the session aging-out function and, at the same time, resource-balancing is performed.

We use Ixia’s IxScriptMate and IxLoad to generate UDP and TCP traffic loads. To experiment how packet size and the number of concurrent session can affect lock contention and performance, we generate 12 different UDP traffic loads varying in frame size of 64, 512 and 1518 bytes in combination of 1,000, 5,000, 10,000 and 65,535 concurrent sessions. Three session table partitioning schemes R2, R4 and R8 representing \( N_{\text{race core}} = 2, 4, \) and 8 are investigated and we test each of the 36 combinations three times to present the averaged result. Lock contents are measured by the in-kernel lock_stat profiler based on the lock_dep infrastructure introduced to the Linux kernel for tracing dependencies between locks. However, lock_stat overhead ranges from 20-30% of system performance and, therefore, the measured throughputs are actually much lower. We present the number of lock contentions and corresponding throughputs in Fig. 3 and Fig. 4 for the different UDP traffic combinations. As can be observed, in the case of 65,536 concurrent sessions, the number of lock contentions can be reduced to 6 and almost 100 times less for 1,518 and 256-byte frames, respectively. And accordingly, the performance is boosted to be at most 3.5 Gbps higher comparing to the global locking scheme. We also note that the number of concurrent sessions does not bring an explicit influence on lock contention and performance. Also, the performance boosts of 256 and 512-byte frames are unobvious (less than 200Mbps) because for small-sized frames, the time spent in connection tracking processing, i.e., the lock hold time, is relatively shorter than that of bigger frames. One anomaly is that there is a large leap in the number of lock contentions in the case of 1518-byte frame with \( N_{\text{race core}} = 4 \).

To testify the effectiveness of the resource balancing algorithm, we use the QuickHTTP test template of IxLoad to generate biased traffic loads, in which each of the four segments is pumped with different numbers of concurrent connections. The resource balancing threshold \( T_{\text{free,conn}} \) is configured to be 0.85. In Test I, the numbers of concurrent connections are 70,000, 30,000, 10,000 and 90,000 for each of the segment, respectively. We simulate the extreme unbalanced case of 195,000, 1,000, 1,000, and 3,000 concurrent connections for each segment in Test II. Several IxLoad performance counters are measured including (1) the number of concurrent connections, (2) connection rate, (3) TCP connection requests failed, (4) TCP lost retransmission, and (5) TCP timeouts. Resource statistics are sampled in one second interval from the kernel module to plot the resource distribution in time series diagrams. In both tests, values of counter (1), (2), and (3) are all zeros during the test period. It demonstrates the effectiveness of the resource balancing algorithm since we adopt the policy to drop packets if no more free connection entry is left and the dropping will cause nonzero values in any of these counters. From Fig. 5, we can see, as the number of concurrent connections increases during the test, unbalances occurs and resource balancing takes effect at the time point of about 121
second which is the same time the numbers of concurrent connections feeding to segment 1 and 4 exceed 55,000. Free connection entries of groups corresponding to segment 2 and 3 are migrated to the group dedicated to segment 1, but then, resources of all the other three groups are migrated to the group dedicated to segment 4 in the time interval from 151 to 166 second. Experimental results of one single heavily-loaded group of CPU cores consuming almost all the resources are shown in Fig. 6. To summarize, the resource balancing algorithm is effective for balancing resources under biased traffic profiles to achieve the same resource capacity as using global un-partitioned connection tracking structures.

V. CONCLUSIONS

In this paper, we propose a session table partitioning scheme to divide the global connection tracking structures into per-CPU localized ones to mitigate the lock contention and cache line bouncing problems. Combined with the resource balancing algorithm, the constraint of fixed-sized partitioning is removed. Experimental results highlight the effectiveness of our method in reducing a maximum of 100 times of lock contentions and, in turn, boosting performance to be 3.5Gbps higher. Moreover, resources are fully utilized under biased traffic profiles to achieve the same capability without session table partitioning.

REFERENCES