High-speed Connection Tracking in Modern Servers

Massimo Girondi Marco Chiesa Tom Barbette

KTH Royal Institute of Technology

***Abstract*—The rise of commodity servers equipped with high-speed network interface cards poses increasing demands on the efficient implementation of connection tracking, *i.e.*, the task of associating the connection identifier of an incoming packet to the state stored for that connection. In this work, we thoroughly investigate and compare the performance obtainable by different implementations of connection tracking using high-speed real traffic traces. Based on a load balancer use case, our results show that connection tracking is an expensive operation, achieving at most 24 Gbps on a single core. Core-sharding and lock-free hash tables emerge as the only suitable multi-thread approaches for enabling 100 Gbps packet processing. In contrast to recent beliefs, we observe that newly proposed techniques to "lazily" delete connection states are not more effective than properly tuned traditional deletion techniques based on timer wheels.**

***Index Terms*—Connection tracking, load balancer, high-speed networking, multi-core processing, hash table**

I. Introduction

Connection tracking is a critical task for many modern high-speed network applications including load balancers, stateful firewalls, network address translators, and general-purpose networking stacks. The goal of connection tracking is to associate the connection identifier of a packet to a *connection state* (*i.e.*, the data that an application stores and uses to process all the packets belonging to a specific connection). It is therefore essential to design the underlying data structures and algorithms used for connection tracking efficiently, so that network applications can operate at multi-gigabits speeds and with minimal latency.

The most common data structures used for connection tracking are hash tables. These data structures efficiently map each input key to its associated value. Different hash table implementations exist and vary in their performance guarantees. The Linux TCP networking stack tracks TCP connections using "chained" hash tables [1]. An eBPF program uses the same type of hash table to share data with user-level programs [2]. The Facebook Katran Load Balancer [3] relies on the eBPF's implementation to map incoming packets to the correct back-end server. CUCKOOSWITCH [4], Krononat [5] and MemC3 [6], use "cuckoo" hash tables [7].

This is the accepted version published on IEEE HPSR 2021. The final version is available at https://ieeexplore.ieee.org/document/9481841, DOI: 10.1109/HPSR52026.2021.9481841

978-1-6654-4005-9/21/$31.00 ©2021 IEEE

The rise of network card speeds up and above 100 Gbps [8–10] poses some unique requirements on the design of the connection tracking component (and its hash tables). We focus on network functions and stacks running on commodity servers, which is the prevalent deployment approach in the emerging SDN/NFV-based modern networks [3, 11–16]. The following properties should be supported: *i) efficient operations* into the table as packets arrive at high-speed networking interface cards in the order of tens (or hundreds) of nanoseconds, *ii) low memory* utilization as efficient packet processing requires to keep the data structures in the memory caches to the largest extent possible, *iii) scale to multiple cores* as a single core cannot cope with all the incoming traffic, and *iv) handling up to millions* of flows, a common requirement for datacenter load balancers [17].

Contributions: In this work, we embark upon a comprehensive measurement study of the performance of different hash table data structures for supporting the above high-speed networking requirements. All our experiments stress these data structures with traffic up to 100 Gbps using real-world traffic traces.

Our work could be divided as follows:

- We first measure the throughput and performance of insertion and lookup operations with five different types of hash tables that run on a single core. We carefully selected some of the most efficient implementations already available in the literature or used in open source projects.
- We then quantify the scalability bottlenecks of six different techniques that can be used to scale a data structure in a multi-core environment.
- We compare different thread-safe hash table implementations with the widely adopted *core sharding* approach [5, 18–21] where each core is assigned its own hash table and the network card sends each incoming packet to the core that has the connection state to process that packet.
- We finally focus on deletion operations on the hash table. Deletion operations are typically performed after the application does not receive a packet of a certain connection for a certain (configurable) amount of time.

We note that deletion operations on general hash tables have received fewer attentions [22–24] than insertion and lookup operations [4, 6, 18]. Deletion operations are however performed *as often as* insertion operations and must therefore be implemented efficiently. Deletions are also key to achieve low memory utilization (by promptly evicting non-active con-

nections flows) and avoid the well-known performance degradation that arises when the number of elements in a hash table approaches it maximum limit. We focus our measurement of deletion techniques on three different mechanisms. In a multi-thread application, the deletion is even more cumbersome: other threads may access the connection table while an entry is deleted, requiring further controls.

We derive the following main findings for any developer seeking to develop a connection tracking application:

- Recently proposed hash tables improve throughput performance by around 15% in multiple scenarios over a traditional "chained" hash table. The performance however sharply degrades as the utilization of the tables get close to its maximum.
- Relatively simple multi-core scaling solutions such as spinlock, mutexes, or hierarchical locking scale extremely poorly, not being able to take advantage of more than three cores for a load balancer use case.
- Core-sharding is the easiest scaling approach for multi-core network applications. Lock-free techniques achieve similar performances of core-sharding at the cost of an increased complexity for deletion operations.
- Traditional deletion techniques based on a "timer wheel" performs equivalently to recently proposed "lazy" deletions without making insertion operations more complex, and scales better. Scanning the table to evict connections should be avoided.

We have published all the code, data, reproducibility tools, and supplementary figures omitted for space constraints (latency, cache misses, ...) on our GitHub repository [25].

II. Data-structures for connection tracking

Connection tracking applications rely on a *connection table* to perform the association between the connection identifier of a packet (*e.g.*, the TCP/IP 5-tuple, a QUIC connection identifier, the GTP TEID of mobile traffic) and the corresponding stored connection state.

As modern network applications must process packets within a very strict time budget to avoid packet queuing (and consequently packet drops), the implementation of the connection table has received much attention in the past. On a 100 Gbps link, the time budget for processing a packet can be in the same order of magnitude of a memory access.[1] It is therefore of paramount importance to keep the connection table small so that it better fits in the CPU memory caches.

Binary trees and hash tables [28, 29] can both be used to implement a connection table. Binary trees offer good performance when used to classify packets based on general filtering policies (possibly including wildcard matches), but suffer from costly updates and inserts. When the classification task is performed on a specific set of fields with an exact match, as it is the case for connection tracking, hash tables provide better performance (*i.e.*, constant-time lookup and average constant-time insertions). We therefore focus only on hash table data structures.

Hash tables. In its easiest form, a hash table stores elements based on an index obtained by *hashing* the key of the element. When two keys are hashed to the same index, a collision happens. The major difference between different hash table implementations is the technique used to handle such collisions. In this work, we focus on the following hash tables:

- *Chained hash tables* handle collisions using an *open hashing* strategy. Each bucket in the hash table stores all colliding elements in a linked list. A lookup operation simply entails accessing the linked list indexed by the hash of the element key and scan all its elements. Chained hash tables may lead to slow lookup operations as linked lists may grow arbitrarily large (containing all the inserted elements in the worst case).
- *Cuckoo hash tables* [7] implement a *closed hashing* approach, where collisions are resolved by assigning two (or more), independent, buckets to a given key. During a colliding insertion, all the buckets for that key are checked and, if all are occupied, a swap process is started. The incoming element is inserted in any of the buckets and the insertion procedure is repeated for the replaced element. The main advantage of this algorithm is the constant-time lookup operation, which is independent of the table utilization. Several improvements have been proposed upon the original schema, such as partial hashing [6], concurrent optimizations [30], and pre-fetching predictions [18, 30]. Cuckoo hash table implementations are broadly available in many data-plane libraries, such as DPDK [31]. We evaluate the implementation offered by DPDK [32], using a traditional design enhanced with buckets that can store up to K elements instead of only one.
- *Cuckoo++* [18] improves the cuckoo baseline by adding a Bloom filter [33] to each bucket. This filter indicates whether a key is certainly not stored in the other possible bucket, saving one unnecessary memory access to that bucket.
- *Hopscotch* [34] is a closed hashing scheme where entries are relocated closely to the original location of a key. It resolves collisions by storing the colliding entries in a *neighborhood* of the initial location. If the neighborhood fits in a cache line, the cost to access any colliding entry will be close to finding an entry in the primary location.
- *Robin-hood* [35] is a closed hashing scheme that relocates entries using a linear-probing approach: when a collision is found, the following adjacent entries are iteratively inspected, swapping the entries if the inspected entry is closer to its original location than the one that needs to be inserted.

All of these implementations behave similarly on a global scale. However, the slightly different implementation details induce distinct performance when employed in a high speed environment.

Multi-core approaches. As networking speeds have grown at a much faster pace than CPU core clock frequencies, handling full line-rate traffic on a single CPU core had become an

[1] On a 100 Gbps link, minimum size packets arrive each 6.72 ns, while a memory access requires around 60 ns [26, 27].

increasingly elusive feat. Consequently, it is paramount to distribute the network application processing among several CPU cores. In connection tracking applications, this typically requires to share the data structure used as connection table. To operate consistently and reliably, the different processes (or threads) of the application must coordinate their read and write operations to keep the table in a consistent state. The main existing approaches to share a data structure are:

- *Lock-based methods* provide mutually exclusive access to a data structure through an explicit synchronization primitive, called a *lock*. A lock on a data structure can be acquired or released by one process and relies on hardware atomic primitives to avoid race conditions. Different types of locks exist that differentiate on how the lock is acquired.
  A *spinlock* is a locking mechanism that performs a busy-wait loop on the lock acquisition operation until it succeeds (*i.e.*, the lock on the resource has been released). This approach minimizes the accessing time for a usually-free resource but it wastes CPU cycles when waiting for the lock.
  A *mutex*, or mutually exclusive lock, is an improvement upon spinlocks. When waiting for a lock to be released, processes are moved to a waiting state, allowing the CPU to schedule other tasks. While this mechanism reduces the CPU usage, the transition from one state to another one introduces additional latency. Both spinlocks and mutex can be acquired either on an entire data structure (*i.e.*, the entire connection table) or on a smaller portion (*e.g.*, per-bucket granularity). In this work, we refer to these fine-grained locking mechanisms as *hierarchical* locks.
- *Lock-free methods* [36, 37] solves a pressing problem of lock-based mechanisms in which a failed process that holds a lock may indefinitely stuck the progress of the other processes. A way to implement lock-free data structures is to keep one or more version counters that are updated whenever the data structure (or some parts of it) are updated. Other processes check the initial and final version number to verify whether their read/write operations were performed correctly or should be corrected (or reverted).
- *Core sharding* is a well-known technique to completely overcome issues related to sharing a resource by assigning a distinct instance of a resource to each process. In modern network cards, RSS [38] distributes incoming packets to each core deterministically based on the connection identifier, *i.e.*, it sends packets with the same connection identifier to the same core using a hash-based load balancer. In the context of connection tracking, each process only stores the connection state for the connections that it receives from RSS, thus eliminating any need to share resources. For some connection tracking applications such as NATs, one may want to guarantee that packets belonging to a connection are delivered to the same core in both directions.

Flow aging and deletions. Handling connection termination is a delicate task in a connection tracking application. Most network protocols are time-based, relying on timing to update their status. For instance, even after sending the last FIN, a TCP connection must wait for a certain amount of time before being closed. Therefore, connection tracking applications most often recognize expired connections using a time interval: if packets have not been seen for a certain connection for a given amount of time, the connection is considered expired and its state can be deleted.

When deleting an entry, especially in a multi-thread scenario, attention should be paid in controlling that other threads are not accessing it, either simultaneously or after the deletion. In particular, the deletion process may interleave with a read-access by another thread: while the deleting thread is proceeding to delete an expired entry, the reading thread would prevent the deletion.[2] Thus, the deletion must be protected with some additional mechanisms, always leaving the table in a consistent state.

In this work, we take three approaches to delete entries from the connection table: a scanning-based, a timer wheel, and lazy-deletion.

- *Scanning-based deletion* is the simplest approach that merely consists in periodic scanning of the connection table: when we find an expired entry, we remove them from the connection table. The ratio between the scan interval I (*i.e.*, how frequently the connection table is parsed) and the timeout t (*i.e.*, what is the maximum age for a flow before being deleted) determines the aggressiveness of the algorithm: lower t will delete flows sooner (possibly incorrectly), while higher I will increase the number of flows that are deleted at a single round and requiring a larger table size.
  Aggregating more deletions at a single round may be more efficient for some connection table implementations: if the deletion requires a lock of the connection table, this may be kept during the entire maintenance process. While all the other packets will be delayed, the deletion can proceed faster thanks to the absence of interleaved accesses, speeding up the entire operation.
- *Timer Wheels* are abstractions to efficiently maintain a set of timers in software, used also in major software projects like the Linux kernel [39–41]. At high level, we associate a timer to every entry in the connection table, triggering the deletion of the entry on expiration. We implement it by keeping a set of *timebuckets* for different time ranges, and we store a connection identifier in a bucket with time range $[T; T + \delta)$ if the connection should be deleted in that time range. At regular time intervals, the timers registered in the current bucket are fired, deleting the corresponding connections.
- *Lazy deletion* follows a similarly proposed approach in Cuckoo++ [18], where an extra `last_seen` field is added to the hash table. The field is updated every time an entry is read from the hash table. During the insertion of a new entry, we detect a collision if the existing resident entry has not yet expired. We evict the expired entry otherwise.

[2] In that case, the entry would not be expired anymore.

III. Evaluation

We now evaluate the performance of different hash table designs using a simple L4 Load Balancer (LB) implemented in FastClick [42], a faster version of the Click Modular Router [43]. We use the different hash table implementations to store the connection identifiers (*i.e.*, the TCP/IP 5-tuple). The data-structures will then return an integer identifier which is used as an index to access the *connection states*. This array, pre-allocated during the start-up, contains the states of every connection. For the load balancer (LB), the state is the selected destination server. While we limit our analysis to a LB, the results are not closely related to the chosen application and could be generally applied to a multitude of other connection tracking application such as middleboxes, networking stacks, and key-value stores.

A. Test methodology

We perform our evaluation using two server-grade machines, interconnected via a 100 Gbps NoviFlow switch. One machine logically acts as a Traffic Generator (both client and server sides) while the other acts as a LB. The first machine is equipped with two Intel® Xeon® Gold 6246R CPUs, 192 GB of RAM and a Mellanox® ConnectX®-5Ex NIC, while the LB machine is equipped with a Intel® Xeon® Gold 6140, 192 GB of RAM and a Mellanox® ConnectX®-5 NIC. The LB receives and transmits the traffic on the same NIC port. FastClick uses DPDK 20.11 to access the network cards, with each thread statically assigned to a physical core, polling one receiving queue. The application runs on the NUMA node where the network card is connected, accessing only local resources on the same node. Tests are repeated 5 times to show the average value and the standard deviation in the figures.

1) Traffic generation: We use FastClick to replay a traffic trace recorded at our campus router (with $\sim$ 200k connections) and, in some experiments, we also rely on traffic traces from CAIDA [44], which has roughly twice the number of connections and represents traffic transiting at a tier-1 router. To achieve 100 Gbps, we replay up to 32 different windows of the trace simultaneously, shifting the IP and port pairs. This method has the advantage of keeping temporal spacing between the arrival of new flows and subsequent packets. Thus, the resulting traffic looks like the connections generated by a campus up to 32 times bigger or an IXP router with higher speed ports.

2) CPU cycle measurement: We measure CPU cycles for different types of operations by performing differences of the TSC counter, whose values are read before and after each operation on the hash table. While this may introduce an additional cost and for each packet processing operation, this is constant across all the analyzed implementations, giving a fair comparison between all methods.

B. Single core performances

We conduct our research on six different hash table implementations, adopting the schemes already introduced in Section II.

- **Chained**: two chained hash table using per-bucket lists to handle colliding entries, which are stored at the head of the list. We report the performance of the implementation distributed with FastClick, and C++'s *unordered_map* implementation from the standard library.
- **Cuckoo**: a cuckoo based hash table offered by the DPDK framework [32]. It improves the original cuckoo idea with partial hashing [6] and pre-fetching of the buckets [30].
- **Cuckoo++**: a cuckoo based hash table optimized with bloom filters [18].
- **HopScotch**: an Open Source hopscotch hash table implementation [45].
- **RobinMap**: an Open Source hash table implementation based of robin-hood hashing [46].

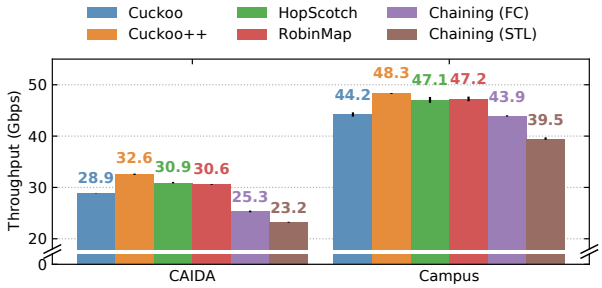

Figure 1. Performance of the 6 methods using a single core, under 2 different traffic scenarios. Both cases shows identical trends, with up to 28% difference in performance.

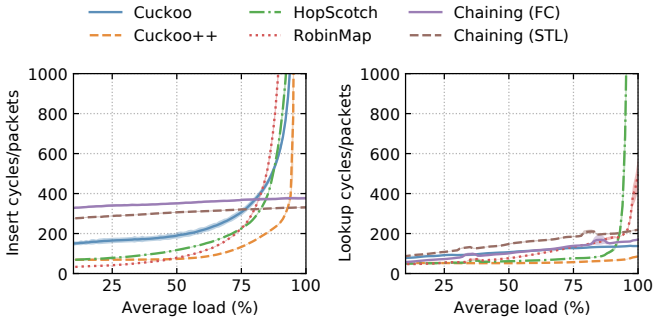

Figure 2. Number of cycles to insert and lookup entries, campus trace 16x, 2M entries. Chained hash table cost increases linearly but performs worst than other methods.

Figure 1 shows the throughput of the LB when processing traffic with only one core using the six implementations with 16 windows of both traces ($\sim$55 Gbps), ensuring the throughput is only limited by the performance of the LB. All hash-tables are provisioned with 4M entries that fit the 10 seconds of trace replayed (as we do not bring up recycling at this stage). The CAIDA traces contain twice more flows than the campus trace, exhibiting around 33% worse performance. However, despite those very different conditions, the trend between the methods is identical: Cuckoo++ performs best, up to $\sim$ 28% better than FastClick's (FC) chaining hash table. The hash table from the C++ standard library (STL) performs

up to $\sim 15\%$ worst than FastClick's due to heavier memory management while the later relies on memory pools and a simpler, more efficient API instead. HopScotch and RobinMap perform similarly, slightly below Cuckoo++. DPDK's Cuckoo implementation performs, averagely, $7 - 10\%$ worse than Cuckoo++.

Figure 2 reports the number of cycles required to process a packet under an increasing table load factors. We do not report load below 10%: at this network speed rate as the load grows to 5% in a fraction of seconds and shows unstable results. While all the implementations slowly increase the number of cycles required to lookup for entries when the tables are becoming full, HopScotch and RobinMap lookup times become unsustainable earlier whereas Cuckoo and Cuckoo++ maintain an almost constant lookup time complexity. Recall that the lookup operation in Cuckoo searches an element in at most two buckets while RobinMap and HopScotch may have to search through more than two adjacent buckets, possibly through the entire table. The number of cycles for Cuckoo and Cuckoo++ lookups slowly increases despite an expected constant complexity: when storing more connections in the table, only a small fraction of them can fit in the cache. We note the chaining method is always more costly up to 80% of capacity, but does not degenerate. The reader should not extrapolate that chaining is a safer solution, the trivia here is that tables should be provisioned big enough to ensure load does not go beyond 75% of their maximum capacity.

C. Multi-core scaling

We study the scaling of the connection table across multiple cores using six methods:

- **Spinlock**: all the access operations to the connection table are protected with a spinlock.
- **Mutex**: readers and writers are protected in a Single-Writer, Multiple-Reader schema. We use C++17 [47] `std::mutex`, acquiring an *exclusive* lock around any write operation on the flow-table. A *shared* lock is instead acquired by the readers.
- **Hierarchical Locking**: we adopt this schema to protect the *chained hash table*, locking either the whole table, the bucket, or the single entry depending on the operation. The STL implementation does not support such mechanism, hence we only study hierarchical locking with the FastClick implementation.
- **Cuckoo LB**: the table is shared among different threads using the *locking* mechanism offered by the *DPDK Cuckoo* hash table implementation. This mechanism offers multiple-reader, single-writer protection using a lock [48] around critical operations on the table, with a strict integration with the hash table code.
- **Cuckoo LF**: the table is shared among different threads using the *lock-free* mechanism offered by the *DPDK Cuckoo* hash table implementation, based on a *version counter* and atomic operations [49, 50], similar to the idea discussed in Section II.
- **Core Sharding**: we duplicate the connection table data structures per each thread, exploiting RSS and assigning one receiving queue to each processor, as discussed in Section II.

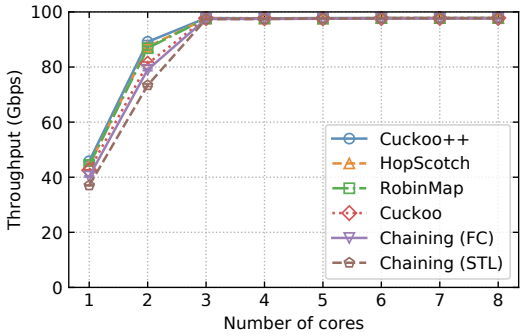

Figure 3. Scaling of the connection tracker using core-sharding: one hash-table per core, campus trace 32x. All the implementations scale linearly until network bottleneck.

Figure 3 shows the throughput for the core-sharding scaling approach and an increasing number of CPU cores used by the load balancer. We can observe how the throughput scales almost linearly for the six implementations using sharding up to 3 cores. From this point, the bottleneck of the system is the 100 Gbps network link, which saturates. To put the overheads of connection tracking into perspective, we note a forwarding configuration achieves 80 Gbps with a single core, and 100 Gbps with 2 cores.

Figure 4 shows the throughput of different locking techniques for an increasing number of CPU cores used by the load balancer. We observe that the performance of lock-based methods sharply degrades when increasing the number of cores. The Cuckoo lock-free is the only one to compete with core-sharding, reaching 100 Gbps and offering an almost-linear scaling. Still, with 2 cores, Cuckoo lock-free delivers 67 Gbps while Cuckoo++ with core-sharding reaches 76 Gbps. The other implementations cannot process more than 61 Gbps even with eight cores, showing the bottleneck is the locking mechanism.

D. Deletion

We then compare the garbage collection techniques presented in Section II in Figure 5, using a single core and the Cuckoo hash table. The load of the CPU is around 70%. The scanning technique is heavy on the CPU cache, and induces an order of magnitude higher number of LLC cache misses. Moreover, even when scanning $1/1000$ of the connection table at a frequency of 1 kHz, the scanning is taking too long and packets are dropped as they accumulate in the receiving queue. We note that at a high enough frequency, the timer wheel mechanism performs similarly to the lazy-deletion. Surprisingly, if the length of the timer wheel buckets does not increase too much (ensured by a frequency high enough), the cycles spent in recycling are equal to the overhead of looking for expired entries and updating the time in the lazy deletion. One could argue that scanning could be performed

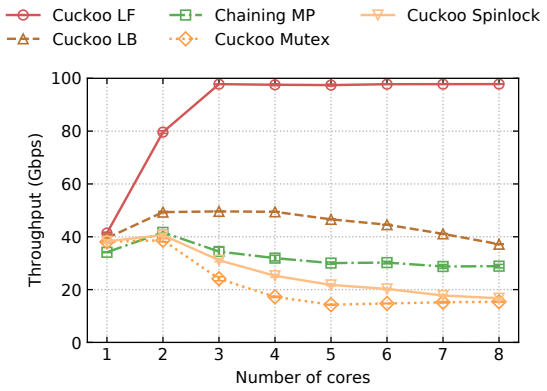

Figure 4. Scaling of connection tracker using locking techniques for a single hash-table, campus trace 32x. Cuckoo Lock-Free scales linearly on the number of cores, while all lock-based mechanisms can achieve an almost constant throughput among cores.

by a remote, eventually dedicated, core. This would waste resources, breaking sharding and forcing costly synchronization mechanisms.

1) Scaling with deletion: When sharding is not possible, deletion becomes more complicated as one thread could recycle entries of other threads. Figure 6 shows the single-table using Cuckoo LF exhibits slightly lower performance than sharding because of the increased contention due to recycling. Similarly, the lazy method do not scale as well as the timer wheel because all threads have to scan buckets for recycled entries which force cache-line transfers.

2) Additional controls implies additional overhead: We also observed that the presence of multi-thread support influences the performance of the system even when using a single core (not shown in a graph). In particular, the Lock-Based approach of DPDK Cuckoo tables reduces the throughput by 10%, similarly to Mutex and Spinlock implementations, even when if there is no thread-race.

IV. Related Work

A. Hash Tables

On top of the methods presented earlier, several works have focused on hash tables optimizations and implementations. CUCKOOSWITCH [4] optimizes the underlying data structures to exploit instruction reordering in x86 CPUs and

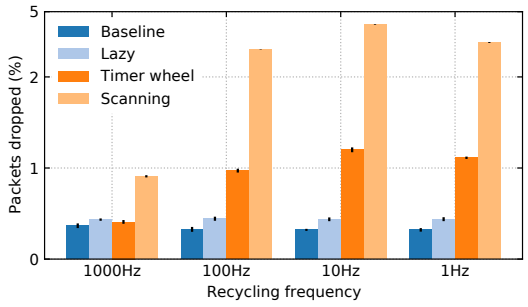

Figure 5. Comparison of deletion methods under an increasing garbage collection frequency running the Cuckoo implementation on a single core. Lazy deletion is comparable to timer wheel at higher recycling frequencies.

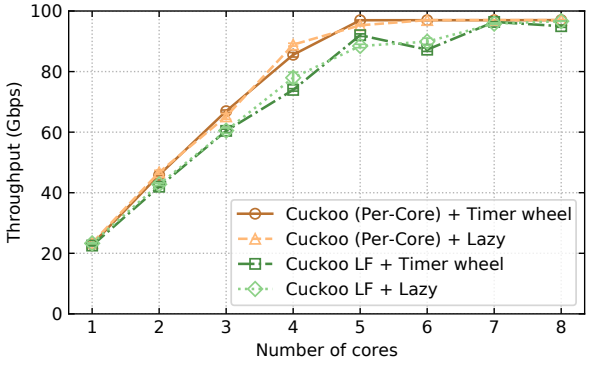

Figure 6. Scaling of two deletion techniques with sharded tables or a single lock-free Cuckoo table, Caida trace 32x. Timer-Wheel per-core scales best, performing up to 20% better than Cuckoo LF using Lazy deletion.

dynamic batching. MemC3 [6] improves Cuckoo Hash tables with algorithmic and work-load based optimizations, some of which have been implemented in the cuckoo implementation used in this work. A stash-memory based solution has been proposed by A. Kirschet et al. [51], demonstrating how a small external area of memory can be used to amortize the cost of collisions and improving the performances at high load factors. Cache Line Hash Tables [52] tightly designs the implementation around the CPU cache structure to reduce the latency of the operations.

Othello [22] and Ludo [23] hashing take a more algorithmic approach, separating the implementation of the hash tables in a control plane and a data plane parts. Concury [53] uses Othello hashing to implement a fast load balancer while Concise [54] implements a packet forwarding mechanism based on Othello, where the control plane part is run on the SDN controller. RUBIK [55] and DUET [56] enhance the packet classification by moving some logic to the ECMP tables of modern switches. Krononat [5] presents a CG-NAT application built on top of [18], using core-sharding to scale for high bandwidths. SilkRoad [17] provides low latency Load Balancing by using ASIC switches, where a digest of the hashes is stored. Cheetah inserts the index value of the entry where the flow state is stored into each packet of a connection, thus relying on simple arrays instead of hash tables [57]. A hardware implementation of hash tables can be realized with Content Addressable Memories (CAMs), which provide a constant and low access time. However, the power consumption and the small capacity limit the scaling of this solution [58]. Recent releases in programmable hardware [59–61], together with recent advancements in this field [17, 62, 63], could open the path to further developments, where connection tracking is offloaded to the network card.

B. Flow Ageing

Iptables implements connection tracking by relying on a Finite State Machine, with ageing regulated by timeouts [64–66]. Cuckoo++ [18] implements ageing with a lazy strategy, similar to our implementation, updating a timestamp when an entry is accessed. However, the authors do not evaluate its efficiency, claiming lazy deletion is supposedly better than

timer wheels, which we could not confirm in our experiments. MemC3 [6] leverages a linked list that stores the last recently used keys. Whenever space is needed, the key at the tail is removed from the table. Bonomi et al. [67] propose to use a flag to distinguish between recently used and expired entries, parsing all the table at regular intervals. Binned Duration Flow Tracking [68] divides the hash tables in a fixed number of ordered bins, across which the flows are swapped. When space is needed, the oldest bin is deleted. Bloom filters are used to speed up the lookup of the flows in the bins. Katran [3] tracks expired UDP connections with a timer, while TCP connections are removed when space is needed in the tables.

C. Future work

Recent releases of NICs with connection tracking mechanisms [9] may open the path to implementations that exploit hardware offloading on commodity hardware, freeing the servers' CPU from the connection tracking duty [69].

To improve the memory efficiency and processing time, the structure and the size of flow table data-structures may be changed at run-time. While static allocation of the structures is usually preferred, limiting the variables that may influence the packet processing time, some works have already explored this possibility [15, 70]. The transformation between different data structure types may result in a more optimized response under some specific loads (*e.g.*, one implementation may optimize insertions in an insert-intensive workload, while another may optimize lookups or deletions). This conversion requires a careful design in order to reduce the packet-processing time during the transition.

V. Conclusions

This work presents an analysis of six different hash tables implementations based on a load balancer application, showing up to 30% differences in performance. Cuckoo++ resulted the most efficient hash table implementation, gaining a 10% higher throughput than a basic Cuckoo implementation and up to 28% better than basic chained hashing.

The need to scale to multiple cores showed that only core-sharding and lock-free connection tables can achieve high-throughput. However, lock-free implementations heavily depend on the workload (*e.g.*, write or read intensive), requiring a careful design to keep the number of cycles low. Core-sharding is the only approach that can truly scale linearly on the number of cores, independently to the offered traffic load. We then explored recycling, an angle generally forgotten by other works. We found that timer wheels, when run frequently enough, compensate the cost of real-time garbage collection, delivering similar performances to lazy deletion. The latter is penalized by the additional cycles required for timestamp comparisons on all operations, achieving slightly lower performance.

Acknowledgments

This project was funded by the Swedish Foundation for Strategic Research (SSF). This project has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889). The work of the second author was partially supported by the KTH Digital Futures center.

References

[1] A. Jaakkola, "Implementation of transmission control protocol in linux," in *Proceedings of Seminar on Network Protocols in Operating Systems*.
[2] Facebook. (2020) eBPF hashmap implementation in the Linux kernel. [Online]. Available: https://git.io/Jt0mF
[3] Facebook Incubator. Katran GitHub Repository. https://github.com/facebookincubator/katran.
[4] D. Zhou *et al.*, "Scalable, high performance ethernet forwarding with cuckooswitch," in *ACM CoNEXT 2013*.
[5] F. André *et al.*, "Don't share, don't lock: large-scale software connection tracking with krononat," in *USENIX ATC 2018*.
[6] B. Fan *et al.*, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *USENIX NSDI 2013*.
[7] R. Pagh *et al.*, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
[8] Intel Barefoot Networks. (2020) Tofino-2 second-generation of world's fastest p4-programmable ethernet switch. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino-2/
[9] NVIDIA Mellanox. (2020) ConnectX®-6 DX EN IC 200GbE ethernet adapter ic. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-connectx-6-dx-en-card.pdf
[10] Z. Yao *et al.* (2016) Introducing backpack: Our second-generation modular open switch. [Online]. Available: https://engineering.fb.com/data-center-engineering/introducing-backpack-our-second-generation-modular-open-switch/
[11] G. P. Katsikas *et al.*, "Metron:{NFV} service chains at the true speed of the underlying hardware," in *USENIX NSDI 2018*.
[12] J. Martins *et al.*, "Clickos and the art of network function virtualization," in *USENIX NSDI 2014*.
[13] S. Palkar *et al.*, "E2: A framework for nfv applications," in *ACM SOSP 2015*.
[14] A. Bremler-Barr *et al.*, "Openbox: a software-defined framework for developing, deploying, and managing network functions," in *ACM SIGCOMM 2016*.
[15] L. Molnár *et al.*, "Dataplane specialization for high-performance openflow software switching," in *ACM SIGCOMM 2016*.
[16] D. E. Eisenbud *et al.*, "Maglev: A Fast and Reliable Software Network Load Balancer," in *USENIX NSDI 2016*.
[17] R. Miao *et al.*, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *ACM SIGCOMM 2017*.
[18] N. Le Scouarnec, "Cuckoo++ hash tables: High-performance hash tables for networking applications," in *ACM ANCS 2018*.
[19] T. Barbette *et al.*, "Rss++: Load and state-aware receive side scaling," in *ACM CoNEXT 2019*.
[20] D. Didona *et al.*, "Size-aware sharding for improving tail latencies in in-memory key-value stores," in *USENIX NSDI 2019*.
[21] S. Han *et al.*, "Megapipe: a new programming interface for scalable network i/o," in *USENIX OSDI 2012*.
[22] Y. Yu *et al.*, "Memory-Efficient and Ultra-Fast Network Lookup and Forwarding Using Othello Hashing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 3, pp. 1151–1164, 2018.
[23] S. Shi *et al.*, "Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.
[24] Z. Chen *et al.*, "Concurrent hash tables on multicore machines: Comparison, evaluation and implications," *Future Generation Computer Systems*, vol. 82, pp. 127 – 141, 2018.
[25] Repository with additional material. [Online]. Available: https://github.com/conntrackHPSR21
[26] L. Barroso *et al.*, "Attack of the killer microseconds," *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, 2017.
[27] A. Farshin *et al.*, "Make the Most out of Last Level Cache in Intel Processors," in *ACM EuroSys 2019*.
[28] P. Gupta *et al.*, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, March-April/2001.

[29] S. S. e. Dinesh P. Mehta, *Handbook of data structures and applications*. Chapman & Hall/CRC, 2004.

[30] X. Li *et al.*, "Algorithmic improvements for fast concurrent cuckoo hashing," in *ACM EuroSys 2014*.

[31] DPDK Project. (2020) Dpdk website. [Online]. Available: https://dpdk.org

[32] DPDK Project. (2020) Hash library. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/hash_lib.html

[33] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970.

[34] M. Herlihy *et al.*, "Hopscotch hashing," in *DISC 2008*.

[35] P. Celis *et al.*, "Robin hood hashing," in *SFCS 1985*.

[36] M. Herlihy *et al.*, *The Art of Multiprocessor Programming (Second Edition)*, 2nd ed. Morgan Kaufmann, 2021.

[37] M. P. Herlihy, "Impossibility and universality results for wait-free synchronization," in *ACM PODC 1988*.

[38] Intel®. (2016) Improving network performance in multi-core systems. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf

[39] G. Varghese *et al.*, "Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility," *IEEE/ACM transactions on networking*, vol. 5, no. 6, pp. 824–834, 1997.

[40] J. Corbet. (2015) Reinventing the timer wheel. [Online]. Available: https://lwn.net/Articles/646950/

[41] J. Corbet. (2016) timer: Refactor the timer wheel. [Online]. Available: https://lwn.net/Articles/691064/

[42] T. Barbette *et al.*, "Fast userspace packet processing," in *ACM ANCS 2015*.

[43] E. Kohler *et al.*, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[44] The CAIDA Anonymized Internet Traces. (2019). [Online]. Available: http://www.caida.org/data/passive/passive_dataset.xml

[45] Hopscotch-map GitHub repository. [Online]. Available: https://github.com/Tessil/hopscotch-map/

[46] robinmap-map GitHub repository. [Online]. Available: https://github.com/Tessil/robin-map/

[47] *ISO/IEC 14882:2017 Information technology - Programming languages - C++*, ISO Std., 2017. [Online]. Available: https://www.iso.org/standard/68564.html

[48] DPDK Project. (2020) Reader-writer lock library documentation. [Online]. Available: https://doc.dpdk.org/api/rte__rwlock_8h.html

[49] H. Nagarahalli. Patch introducing address reader-writer concurrency in rte_hash. [Online]. Available: https://mails.dpdk.org/archives/dev/2018-September/111016.html

[50] H. Nagarahalli. Lock Free RW Concurrency in hash library. [Online]. Available: https://www.dpdk.org/wp-content/uploads/sites/35/2018/10/am-04-lock_free_rte_hash_Honnappa.pdf

[51] A. Kirsch *et al.*, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal on Computing*, vol. 39, no. 4, p. 1543–1561, 2009.

[52] T. David *et al.*, "Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures," in *ACM ASPLOS 2015*.

[53] S. Shi *et al.*, "Concury: a fast and light-weight software cloud load balancer," in *SoCC 2020*.

[54] Y. Yu *et al.*, "A concise forwarding information base for scalable and fast name lookups," in *IEEE ICNP 2017*, 2017.

[55] R. Gandhi *et al.*, "Rubik: Unlocking the Power of Locality and End-Point Flexibility in Cloud Scale Load Balancing," in *USENIX ATC 2015*.

[56] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," in *SIGCOMM 2014*.

[57] T. Barbette *et al.*, "A high-speed load-balancer design with guaranteed per-connection-consistency," in *USENIX NSDI 2020*, pp. 667–683.

[58] F. Baboescu *et al.*, "Packet classification for core routers: Is there an alternative to CAMs?" in *IEEE INFOCOM 2003*.

[59] Netronome. (2019) Agilio fx smartnic. [Online]. Available: https://www.netronome.com/products/agilio-fx/

[60] Netronome. (2019) Agilio cx smartnic. [Online]. Available: https://www.netronome.com/products/agilio-cx/

[61] NVIDIA Mellanox. (2020) Bluefield 2 SmartNIC. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-bluefield-2-smart-nic-eth.pdf

[62] S. Pontarelli *et al.*, "FlowBlaze: Stateful Packet Processing in Hardware," in *USENIX NSDI 2019*.

[63] D. Firestone *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *USENIX NSDI 2018*.

[64] M. Boye. Netfilter connection tracking and nat implementation. [Online]. Available: https://wiki.aalto.fi/download/attachments/69901948/netfilter-paper.pdf

[65] A. Chiao. Connection tracking (conntrack): Design and implementation inside linux kernel. [Online]. Available: https://arthurchiao.art/blog/conntrack-design-and-implementation/#connection-tracking-conntrack

[66] How long does conntrack remember a connection? [Online]. Available: https://unix.stackexchange.com/a/524320

[67] F. Bonomi *et al.*, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," in *ACM SIGCOMM 2016*.

[68] B. Whitehead *et al.*, "Tracking per-flow state—binned duration flow tracking," in *IEEE SPECTS 2020*.

[69] G. P. Katsikas *et al.*, "What you need to know about (smart) network interface cards," in *PAM 2021*.

[70] T. Barbette *et al.*, "Building a chain of high-speed VNFs in no time," in *IEEE HPSR 2018*.