Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing

Chaoliang Zeng[1*] Layong Luo[2] Teng Zhang[2] Zilong Wang[1*] Luyang Li[3*] Wenchen Han[4*]
Nan Chen[2] Lebing Wan[2] Lichao Liu[2] Zhipeng Ding[2] Xiongfei Geng[2] Tao Feng[2]
Feng Ning[2] Kai Chen[1] Chuanxiong Guo[2]
[1]*Hong Kong University of Science and Technology* [2]*ByteDance* [3]*ICT/CAS* [4]*Peking University*

Abstract

Stateful layer-4 load balancers (LB) are deployed at datacenter boundaries to distribute Internet traffic to backend real servers. To steer terabits per second traffic, traditional software LBs scale out with many expensive servers. Recent switch-accelerated LBs scale up efficiently, but fail to offload a massive number of concurrent flows into limited on-chip SRAMs.

This paper presents Tiara, a hardware architecture for stateful layer-4 LBs that aims to support a high traffic rate (> 1 Tbps), a large number of concurrent flows (> 10M), and many new connections per second (> 1M), without any assumption on traffic patterns. The three-tier architecture of Tiara makes the best use of heterogeneous hardware for stateful LBs, including a programmable switch and FPGAs for the fast path and x86 servers for the slow path. The core idea of Tiara is to divide the LB fast path into a memory-intensive task (*real server selection*) and a throughput-intensive task (*packet encap/decap*), and map them into the most suitable hardware, respectively (i.e., map *real server selection* into FPGA with large high-bandwidth memory (HBM) and *packet encap/decap* into a high-throughput programmable switch). We have implemented a fully functional Tiara prototype, and experiments show that Tiara can achieve extremely high performance (1.6 Tbps throughput, 80M concurrent flows, 1.8M new connections per second, and less than 4 us latency in the fast path) in a holistic server equipped with 8 FPGA cards, with high cost, energy, and space efficiency.

1 Introduction

Large service providers deploy various services inside their geo-distributed datacenters of different scales. At the boundary of these datacenters, stateful layer-4 load balancers (LB), a.k.a., multiplexers (Mux), are deployed to distribute user requests from the Internet to many real servers inside datacenters while preserving connection consistency. Driven by exponentially increased content delivery and cloud computing demands, a typical LB in large service providers usually has to process terabits per second of Internet traffic, with tens of millions of concurrent flows [25, 31] and millions of new connections per second (CPS) [12].

To support such high performance, vendor-proprietary hardware LBs (e.g., F5 [9]) were deployed in the early days of some datacenters. However, they lacked agility, which is highly desirable in modern hyper-scale datacenters. In recent years, the move from vendor-proprietary hardware to in-house software LBs, or software Muxes (SMux), e.g., Ananta [36] and Maglev [21], was mainly driven by requirements like manageability, reliability, and agility, but sacrificed efficiency (i.e., cost, energy, and space efficiency). For example, Ananta [36] achieves 10 Gbps per instance, and supporting up to terabits per second throughput requires scale-out with a large number of servers. Deploying so many servers for just LB is not only costly but also challenging at energy- or space-limited boundaries of massive small/medium-scale datacenters (e.g., 10s-100s of servers in PoPs [15] or edge [40]). Moreover, software LBs usually suffer from high latency and jitter, sometimes comparable to Internet access latency (in the order of milliseconds [24]) when CPU load is high. Such latency churn will adversely impact users' network experience.

To improve the efficiency of software LBs without sacrificing agility, there is an emerging trend to accelerate software LBs with in-house software and hardware co-design. Recent work [16, 23, 24, 31] leverages programmable switches to accelerate LBs. Nevertheless, programmable switches have inherent scalability issues (§2.3). *On the data plane*, a modern switch cannot store a large number of concurrent flows due to its small memory size (typically 50-100 MB on-chip SRAMs); *on the control plane*, the switch fails to support a large CPS given its slow entry insertion speed ($\sim$ 100 Kps).

Existing switch-accelerated LBs do not address both challenges simultaneously. For example, Silkroad [31] stores a small hash of a connection instead of the 5-tuple to compress the connection table. However, its scalability is still bounded by the switch's small memory size, and it may suffer from

[*] This work is done while Chaoliang Zeng, Zilong Wang, Luyang Li, and Wenchen Han are interns in Bytedance.

throughput reduction due to switch pipeline folding. Moreover, Silkroad does not address the scalability problem on the control plane. Cheetah [16] provides a fast entry insertion mechanism by storing an index in the packet header but requires modifications on services' client sides. Thus, applying such a mechanism is difficult, if not impossible, in the datacenter with thousands of services [19, 36]. Furthermore, it does not address the scalability issue on the data plane.

One plausible approach to address the above problems is to leverage traffic locality in hardware offloading. If the traffic pattern follows a long-tail distribution (i.e., a small number of flows carry the majority of the traffic), only a few elephant flows need to be offloaded and stored in the switch, thus lowering the requirements of both the hardware memory size and entry insertion speed. However, we observe from production datacenters that the traffic patterns at datacenter boundaries do not necessarily follow a long-tail distribution. Instead, the mix of VIP traffic for multiple services is highly dynamic and unpredictable, detailed in §2.2.

Based on the above analysis and observation, we ask: *can we design a scalable and efficient stateful LB without any assumption on traffic patterns?* Specifically, the design should be:

- **scalable** on both data plane (store > 10M concurrent flows) and control plane (support > 1M CPS);
- **efficient** in terms of high cost, energy, and space efficiency; and
- **generic** without any assumption on traffic patterns.

To this end, we move one step further beyond the existing *switch-server* architecture [23, 24] by exploring more flexible hardware, i.e., FPGA. FPGA is a high-performance and programmable device becoming an important building block in the datacenter infrastructure [22, 28, 29, 42]. The modern FPGA equipped with gigabytes of high-bandwidth memory (HBM) is well-suited to improve LB scalability, as HBM can store a large number of concurrent flows with high lookup and insertion rate.

In this paper, we present Tiara, a three-tier hardware acceleration architecture composed of a programmable switch, FPGAs, and commodity servers, for a high-performance stateful LB with scalability and efficiency. The core idea behind Tiara is that we map different LB tasks into different devices by matching task requirements with device capabilities (§3.1). Specifically, Tiara divides the LB fast path into *real server selection*, a memory-intensive task with both large capacity and high bandwidth requirements, and *packet encap/decap*, a throughput-intensive task. Then, Tiara maps these two tasks into FPGAs with large HBM and a programmable switch with high packet processing throughput, respectively. Similar to other hardware-accelerated systems [23, 24, 37], Tiara leverages commodity servers as the slow path to handle the unprocessed traffic from the fast path.

To support high CPS without compromising line-rate packet processing in a heterogeneous system, we optimize several key design components in Tiara (§3.3). First, for both fast lookup and insertion, Tiara adopts fixed-length hash chaining, which leverages the parallel processing capability in both FPGAs and multi-core servers. Second, we design a lock-free offloading approach to support issuing millions of entry operations per second from a server to an FPGA. Third, Tiara employs a lightweight aging mechanism to recycle outdated entries, where FPGAs periodically report connection activeness via a dedicated accessing bitmap, preventing interference with the data plane.

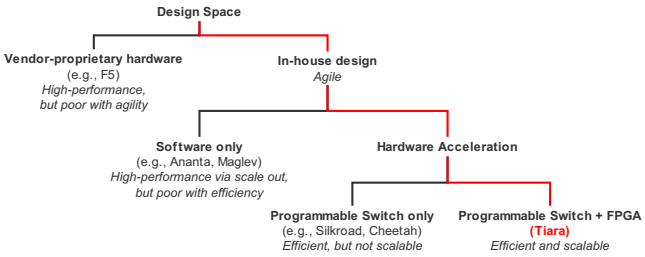

Figure 1: Design space for stateful LB architectures.

We have implemented a fully functional Tiara prototype based on a Barefoot Tofino switch, a Xilinx FPGA-based SmartNIC card, and a commodity server. We modified a production-level SMux for the slow path and the control plane (§4). The key results from our experiments (§5) show that our prototype can support 10M concurrent flows and 1.8M CPS, 9× better than Silkroad [31], at 200 Gbps with less than 4 us average latency and small jitter in the fast path. In a holistic server with 8 FPGA cards, Tiara can provide superiority in throughput (up to 1.6 Tbps) and flow capacity (up to 80M concurrent flows). Meanwhile, Tiara achieves 17.4×, 12.8×, and 16.8× higher cost, energy, and space efficiency, respectively, compared to SMux.

As a summary, Figure 1 shows the design space for stateful LB architectures and the unique position of Tiara. Tiara is more agile than traditional vendor-proprietary hardware, faster and more cost-, energy-, and space-efficient than software LBs, and more scalable than switch-accelerated solutions. Specifically, Tiara makes the following contributions:

- We propose a three-tier architecture that matches key LB tasks to the most suitable hardware: programmable switch for packet encap/decap, FPGA with HBM for connection management, and x86 CPU for SMux.
- We design and optimize key LB components, including an efficient hash table structure for fast lookup and efficient insertion, a lock-free offloading approach to improve connection offloading speed, and a lightweight aging mechanism with little overhead and minimal interference on the data plane.
- We implement the Tiara prototype and conduct testbed experiments to show its performance superiority.

2 Background

2.1 Layer-4 Load Balancing

Layer-4 LB can be classified into the stateful LB, which stores the connection-to-real server (RS) mapping as a connection table (CT), and the stateless LB, which does not maintain any per-connection state. Most of the industry LBs are stateful [3, 5, 8, 21, 36] because stateful LBs can easily ensure *per connection consistency (PCC)* [16, 31], which means all packets of a connection should be delivered to the same RS to avoid breaking the connection. In this paper, we focus on the stateful LB, which usually contains the following two parts.

Real server selection: The LB selects an RS for each incoming packet by identifying its connection via the 5-tuple in the packet header. The LB selects RS in two ways. For the first packet of a connection, the LB selects an RS based on a pre-defined algorithm, e.g., hash, round-robin, or least-loaded, and creates a connection entry in the CT to record this selection. The LB selects the same RS for the other packets of this connection by looking up the CT. This mechanism ensures PCC. An RS can be specified by a tuple of `{RS_IP, RS_Port}` based on backend service implementations.

Packet encap/decap: After an RS is selected for an inbound packet, the LB encapsulates the packet with `RS_IP` and `RS_Port`. The encapsulation process may include multiple steps in practice. Given a tuple of `{RS_IP, RS_Port}`, the LB enforces Port NAT (`virtual Port` (VPort) $\rightarrow$ `RS_Port`), IP NAT (`virtual IP` (VIP) $\rightarrow$ `RS_IP`), and packet encapsulation with VxLAN. Unlike inbound traffic processing involving both RS selection and packet encapsulation, outbound traffic processing only needs packet decapsulation.

2.2 Nature of Internet traffic at the Datacenter Boundary

Large service providers usually deploy many Internet services in a datacenter, and the Internet traffic at the datacenter boundary is a mix of multiple services' traffic, with the following properties.

The flow distribution of individual services varies. The distribution of service traffic depends heavily on the service's client- and server-side implementations. For example, certain service clients may split an elephant flow into multiple smaller ones to reduce the cost of TCP disconnection over unstable Internet, leading to a uniform distribution. In contrast, other service clients may use short connections for synchronization and long connections for massive data transmission, leading to a long-tail distribution. To show this fact, we analyze flow distributions for three different services, as shown in Figure 2. These three services have various flow distributions: service C shows a uniform distribution (where top 10% flows carry 19.6% traffic), while service A and B exhibit traffic locality

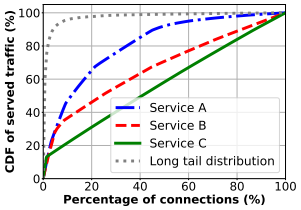

Figure 2: The traffic distribution varies among three different services.

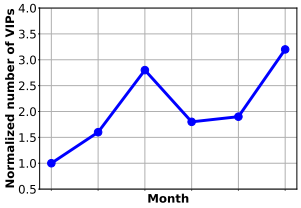

Figure 3: The number of VIPs in a typical LB. It shows a high variation over 6 months.

(where top 10% flows carry 46.3% and 35.5% traffic, respectively) to different extents.

The traffic volume of a service can dynamically change. The traffic volume of an individual service keeps changing independently, with different short-term daily peaks and troughs [21] and long-term uncertainty due to the change in user interest [20]. At any given time in the mixed service traffic, mice flows of one service at peak might consume more bandwidth than elephant flows of another service at the trough, making the overall distribution of their mix unpredictable.

The number of VIPs at a datacenter boundary can change over time. Large service providers keep launching, stopping, and migrating services, driven by various reasons like changes in user interest or business opportunities. Figure 3 reveals a high variation (3.2×) of the number of VIPs served by an LB over 6 months. The dynamic change of services inside the datacenter further makes the mixed VIP traffic at the boundary highly dynamic without any specific distribution.

Based on these observations, we should not rely on any assumption of specific traffic distributions (e.g., long-tail distribution) when designing load balancers at datacenter boundaries for mixed services.

2.3 Accelerating LB with Programmable Switches

Most recent proposals accelerate LBs by realizing hardware Muxes (HMux) [23, 24, 31] with programmable switches, where the RS selection and packet encapsulation are implemented in switch processing pipelines. HMuxes can effectively reduce the number of required servers, which is significant, especially for small/medium-scale datacenters. Nevertheless, using programmable switches as HMuxes suffers from scalability issues on both data and control planes.

Data plane: As widely discussed, switching ASICs cannot support many concurrent flows due to their limited memory sizes [24, 25, 31, 37]. Considering a CT with an entry size of 64 bytes[1] and a typical concurrent flow number of 10M [25, 31],

[1] 64 bytes/entry is an empirical value for IPv6, including 37 bytes for the

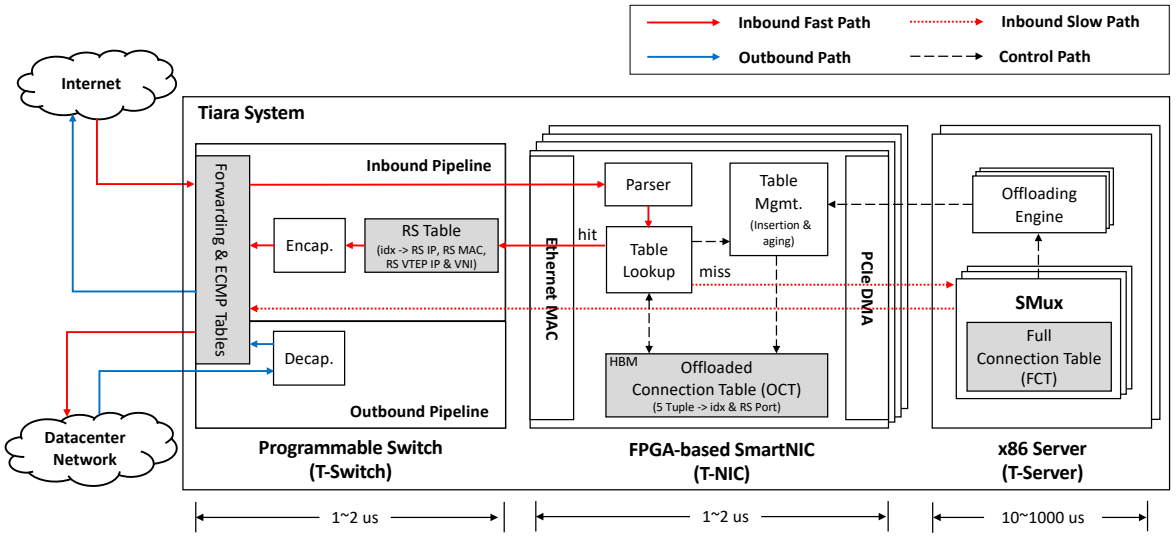

Figure 4: Tiara architecture. Tiara consists of three tiers: T-switch, T-NIC, and T-server. Tiara divides LB into multiple key tasks and matches them respectively to suitable hardware tiers: T-switch for stateless packet encap/decap, T-NIC with HBM for connection lookup and management, and T-server as a last resort.

the CT size is 640 MB. However, modern programmable switches only provide 50-100 MB SRAMs [31]. Moreover, these SRAMs are typically distributed into multiple pipelines, e.g., 15 MB/pipeline. To look up a larger table than a single pipeline's SRAM size, HMuxes typically use folded pipelines and resubmit a packet to switch pipelines via different physical ports, reducing the available throughput.

Control plane: State-of-the-art programmable switches are slow for entry insertion. For example, a Barefoot Tofino switch can only do $\sim$ 100K insertions per second after our optimizations. We measure the entry insertion overhead. Our result reveals that the top two time-consuming functions are the hash computation and the Cuckoo search algorithm [34]. Our result is similar to those of previous work [16, 31]. The root causes exist in the low-end switch CPU, slow PCIe interconnect between the CPU and the switching ASIC, and the small memory size in the switching ASIC. The first two factors affect the speed of hash computation and operation offloading. Then the limited memory space forces the switching ASIC to rely on space-efficient Cuckoo hashing for hash collision resolution. The Cuckoo hashing impedes fast insertion by (1) multiple entry movements during collision resolution and (2) incapability of parallelization due to the dependency between two insertions (the previous insertion location may affect the latter one). The above hardware constraints make it difficult for a switch to support > 1M CPS required by production LBs [5].

3 Tiara Design

We now present Tiara, a novel hardware-accelerated LB architecture, which can support > 1 Tbps traffic, > 10M concurrent flows, and > 1M CPS, without any assumption on traffic patterns.

3.1 Architecture Overview

Tiara is a three-tier architecture as demonstrated in Figure 4. The outermost tier is a programmable switch (T-switch), which sits between the Internet and the datacenter network as a bump in the wire. The second tier is a group of FPGA-based SmartNICs (T-NIC), which act as the HMux jointly with T-switch for LB fast path. The third tier consists of commodity servers (T-server), which host T-NICs and implement SMuxes for LB slow path. The number of T-NICs hosted by a T-server and the number of T-servers behind T-switch are configurable, making the three-tier architecture flexible enough to meet different performance requirements at various datacenter entrances.

The novel idea of Tiara is that it maps different LB tasks into their most suitable devices based on their unique capabilities. In the fast path, Tiara divides the HMux between T-NICs and T-switch. Tiara leverages the large and fast HBM inside T-NIC's FPGA for memory-intensive *RS selection*. One typical HBM stack comprises 16 256-MB memory channels, and each channel provides $\sim$100 million lookups per second (MLPS)[2]. To maximize the accessing performance, we should separate memory accesses to different memory channels. The

5-tuple as match key, 18 bytes for `RS_IP` and `RS_Port` as action data, and a few bytes for packing and alignment overhead.

[2]One memory channel provides $\sim$100 million random read accesses per second based on our emulation [1].

parallelism among HBM channels is carefully explored to meet memory capacity and throughput requirements of RS selection, which will be discussed in §3.3.1. Meanwhile, Tiara leverages the high performance and programmability properties of T-switch pipelines for throughput-intensive *packet encap/decap*.

Besides the fast path processing, Tiara instantiates several SMuxes in T-server to act as a backstop for unprocessed traffic. Each SMux maintains a full connection table (FCT) for all inbound flows, and is associated with a T-NIC virtual function with dedicated DMA channels used for packet receiving and sending.

Programmable switch or fixed-function switch. Another option of Tiara's three-tier architecture demonstrated in Figure 4 is that the programmable T-switch could be replaced by a fixed-function switch that only performs forwarding and ECMP routing. If so, the switch packet processing logic, including RS table, packet encapsulation, and decapsulation, can be moved into T-NICs. We do not choose this option for a few reasons. First, the performance, cost, and power consumption of programmable switches is comparable to that of traditional fixed-function switches [14]. Second, with packet decapsulation implemented in programmable T-switch, the architecture allows outbound traffic to bypass T-NICs (as described in §3.2.2), thus halving the T-NICs bandwidth requirements and the number of required T-NICs. Third, the programmability of T-switch relieves T-NIC implementation. If all fast path functions are implemented in T-NIC, it will increase not only the FPGA size, power consumption, and cost, but also the development time, as programming switches with P4 is easier than programming FPGA with Verilog.

3.2 Control & Data Planes

3.2.1 Control Plane

A typical LB usually includes a centralized controller configuring `VIP` $\rightarrow$ `RS_IP` mappings into Muxes and BGP speakers for VIP announcements. As they are common and well described in previous work [21, 23, 24, 36], we will skip them in this paper and pay more attention to the acceleration-related control flow, i.e., the connection management between software and hardware. Tiara relies on T-servers to make the local control plane decisions, including the CT entry insertion and the entry recycling (connection aging). The powerful CPU prevents inefficient hash computations like that on the switch-based HMux. T-servers use *offloading engines* to offload the entry operations generated by SMuxes, to a specific T-NIC, and each offloading engine is associated with a dedicated DMA channel for entry operations. To efficiently process entry insertion and aging, a few optimizations are made in offloading engines, which will be discussed in §3.3.

Moreover, Tiara integrates many more features like management, telemetry, and fault tolerance in the control plane. Except for the telemetry, Tiara can support all these functionalities solely in the control plane. Network telemetry requires collecting statistic counters from the data plane, and T-NIC and T-switch can provide them easily without affecting the fast path performance.

3.2.2 Data Plane

Inbound fast path. Upon receiving a packet from the Internet, T-switch distributes it to one of the T-NICs based on ECMP. Then, T-NIC parses the packet header and uses the extracted fields (i.e., 5-tuple) to look up the offloaded connection table (OCT), which maintains up to tens of millions of connections in FPGA HBM and sustains fast lookup. The lookup result from OCT is an LB decision, i.e., a two-tuple `{RS_Index, RS_Port}`, where `RS_Index` represents a real server and will be used in later RS table lookup in T-switch. Instead of replacing the `RS_Port` locally, which will incur checksum computation, Tiara delays this operation to T-switch processing. T-NIC encapsulates the retrieved tuple into a packet metadata header between the Ethernet header and the IP header, and sends back the packet to T-switch. T-switch looks up an RS table and gets the corresponding RS information, including `RS_VTEP_IP`, `RS_MAC`, `RS_IP`, and `VNI`. Finally, T-switch enforces Port NAT, IP NAT, and VxLAN encapsulation sequentially, and forwards the encapsulated packet to the RS. Since we decouple the `RS_Port` from the RS table, the number of entries in the RS table is the same as the number of real servers, typically 10K-100K[3]. Compared to CT, the RS table is relatively stable, updated in second time granularity [36], which is far slower than the entry insertion speed provided by T-switch. Based on these two features, the RS table is achievable in the T-switch SRAMs.

Inbound slow path. When a packet misses in the OCT, the T-NIC uploads it to an SMux via a PCIe DMA channel chosen by *Receive Side Scaling* (RSS). Upon receiving the packet, the SMux looks up the FCT and moves to one of the following two workflows according to the lookup result.

If the packet belongs to an established connection, it will hit in the FCT lookup. SMux retrieves the corresponding `{RS_Index, RS_Port}`, and further processes the encapsulation for this packet by looking up the RS table locally[4]. Finally, SMux sends the encapsulated packet to the real server (via T-NIC and T-switch). There is a trick on VxLAN source port calculation. Since the source port is calculated by hashing [13], SMux reuses the last 2 bytes of RSS hash value from the T-NIC to avoid duplicate hash computation.

If it is the first packet of a new connection, it will miss in the FCT lookup. SMux makes the LB decision to create a connection entry for this connection and inserts the generated

[3] A typical datacenter supports thousands of services [19, 36], and each one usually holds 10-100 instances.

[4] In fact, these two tables can fuse into one table.

entry into the FCT. Then, SMux encapsulates the packet and sends it out.

In both cases, SMux will try to insert the corresponding connection entry into OCT. If there are empty slots in the corresponding hash bucket in OCT, the insertion will be successful; otherwise, Tiara will fail the insertion without cache eviction and keep that flow in SMux. We leave the cache eviction policy for the LB connection table as future work.

Outbound path. For outgoing packets, real servers leverage XDP [4] or OVS *Conntrack* [10] to perform SNAT locally. The real servers rewrite source IP with `VIP` and source ports with `VPort`, and forward the packets in VxLAN encapsulation to T-switch. T-switch further performs packet decapsulation and sends the packets to the Internet. As the only LB operation (i.e., packet decapsulation) for outbound packets is offloaded completely in T-switch, outbound traffic can bypass T-NICs and SMuxes, halving the T-NICs bandwidth requirements.

3.3 Component Design & Optimization

3.3.1 Efficient Hash Table Structure

The hash table design of OCT affects not only lookup performance in hardware but also entry insertion speed in software. We leverage an efficient hash table structure that enables both fast lookup in T-NIC and fast entry insertion in T-server. Specifically, we expect the hash table used in T-NIC should (1) support O(1) and parallel insertions in software and (2) support line-rate lookup in hardware.

We observe that a hash table with fixed-length chaining can satisfy all requirements. First, the insertion complexity of hash chaining is O(1). Second, since the hash computations of different insertion indexes are independent, we can utilize multiple cores in T-server to compute the insertion indexes in a parallel manner. Third, T-NIC can support O(1) lookup by mapping fix-length chains into multiple HBM channels. Last, fix-length hash chaining simplifies hardware design. If using variable-length hash chaining, dynamic memory management is mandatory and unfriendly to hardware implementation.

T-NIC manages OCT using a hash table with fixed-length chaining, as illustrated in Figure 5. Despite the simple structure, determining the proper parameters of the hash table in HBM to achieve both fast lookup and low collision rate is non-trivial. For the hash table with fixed-length chaining, two parameters control the shape of the table: the number of hash indexes (*depth*) and the number of entries at each index (*width*), following that $depth \times width = hash\ table\ size$. Given a fixed hash table size, a deeper hash table results in a higher hash collision rate (see analysis in Appendix A), while a wider hash table poses challenges for line-rate lookup on HBM, as the number of parallel HBM memory channels is limited.

Based on the above analysis, T-NIC determines the hash table parameters with a principle that *maximizing the width*

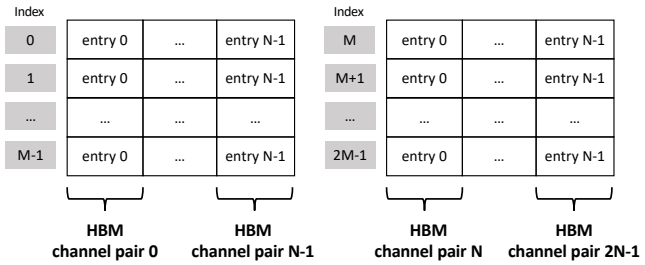

Figure 5: The fixed-length hash chaining design in Tiara OCT, where depth is M and width is N. Each channel pair saves a column of the hash table. For a table lookup, T-NIC can launch multiple parallel accesses of N entries inside $2N$ HBM channels.

while guaranteeing line-rate lookup. Take the FPGA card used in our implementation (§4) as an example. It has two 100GE ports, each requiring 150 MLPS to sustain line rate, and one HBM stack of 16 256-MB (8M × 256-bit width) memory channels, each providing up to 100 MLPS. We divide 16 channels evenly between two ports so that there are 8 channels to support 100 Gbps traffic. Moreover, the entry size is 512 bits, as discussed in §2.3, so we need to pair two channels for one entry access and construct 4 channel pairs for each port. Given that each channel pair can support 8M entries, there are three candidate hash table structures: 8M (depth) ×4 (width), 16M×2, 32M×1, where one lookup operation involves 4, 2, and 1 channel pair(s), respectively. However, the lookup performance of the 8M×4 hash table structure is only 100 MLPS (using all channels for one lookup), failing to support the 100 Gbps line rate. Based on the principle, the best hash table structure for one 100GE port is 16M×2 in our FPGA card.

T-NIC relies on the connected T-server to simplify hash collision resolution. When there is a hash collision in the table lookup, T-NIC will forward the packet to the slow path in T-server; when there is a hash collision in the entry insertion, the insertion fails in the offloading engine (§3.3.2), and that flow will be kept in the slow path. As long as the hash collision rate is low (2.6% in theory for 10M flows in the 16M×2 hash table), hash collision does not have significant performance penalty.

3.3.2 Lock-free Offloading Approach

We design a lock-free offloading approach to enable issuing millions of insertion or deletion operations per second from SMuxes to T-NIC, which is required to support > 1M CPS.

In Tiara, SMuxes offload the generated entry operations, including entry insertion and deletion, to T-NICs via offloading engines. Given the multi-channel feature of our PCIe DMA engine, Tiara instantiates a few offloading engines and associates each with a dedicated DMA channel, so that offloading engines can offload entries independently.

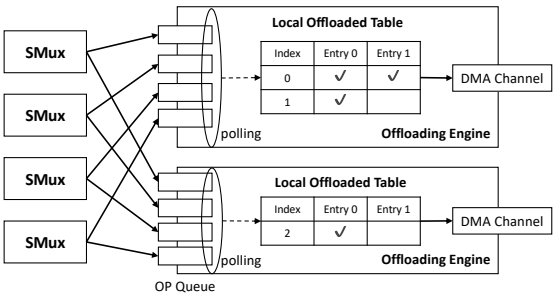

Figure 6: Tiara's lock-free offloading design.

A straightforward offloading approach introduces locks in two places. The first lock happens when multiple SMuxes are mapped to the same offloading engine with only one OP queue. SMuxes can write their operations to the OP queue only when they retrieve a write lock. The second lock exists when multiple offloading engines insert entries into the same hash index. A lock is required for unavoidable synchronizations on a global OCT, maintained in the server to track the OCT usage among different offloading engines. These two locks prevent us from fully leveraging the parallelism in both the multi-core server and the multi-channel DMA to achieve fast entry offloading.

We design a lock-free offloading mechanism, as shown in Figure 6. First, to realize lock-free entry delivery from SMux to the offloading engine, Tiara sets up an OP queue for each SMux-engine pair. The offloading engine polls OP queues in a round-robin manner to retrieve the offloading operations. Second, Tiara adopts the mapping method based on the entry's hash index, i.e., *index-to-engine* mapping. Entries inserted into the same place are delivered to the same offloading engine. Consequently, different offloading engines handle entries with different hash indexes, and each offloading engine maintains a local OCT to track the offloaded indexes. Since the local OCTs are disjoint with each other, it is lock-free during the table update.

For each entry operation, offloading engines will notify SMuxes whether the operation is successful or not (an insertion will fail when the corresponding hash bucket is full) via completion queues (not shown in Figure 6).

3.3.3 Lightweight Aging Mechanism

The purpose of this component is to recycle outdated entries in the OCT, i.e., when a connection is disconnected, its related entry in the OCT should be released so that it can be reused for new connections. To realize it, we need to detect the close of connections. One naive method is to use the TCP FIN packet as the signal of the connection close, which can be captured in T-NICs. However, this method fails on abnormal close of TCP traffic and connection-free UDP traffic.

To unify the flow removing process for TCP and UDP, Tiara adopts an entry aging mechanism that removes a flow entry from the OCT if it is not accessed in a period T. This aging mechanism may kick out connections whose access interval is larger than T by mistake, but those connections can be further processed in the slow path FCT[5].

The challenge of this aging mechanism is to monitor the accessing states of 10M connection entries periodically with a small memory footprint, minimal performance interference on the data plane, and low CPU overhead.

To address this challenge, T-NIC leverages an accessing bitmap to track connection activities, signals activities to SMuxes, and SMuxes make aging decisions by issuing entry deletion operations based on signals.

T-NIC maintains the accessing bitmap in on-chip SRAMs, using each bit as an indicator for a connection entry. All indicators are reset to 0 at the beginning of every detection period Δ_t ($< T$). An indicator will be marked as active, i.e., set to 1, only if a packet is accessing the corresponding connection entry. As an active signal, the packet header will be sent to an SMux by RSS, ensuring that the same SMux processes both teardown and establishment for a connection. Subsequent packets accessing active connection entries neither change the indicator status nor trigger signaling to SMuxes. In this way, if the connection is active in a detection period, the related SMux will get a signal. If the SMux does not receive any signal for a connection in multiple continuous (T/Δ_t) periods, that connection is considered outdated and should be aged. T-NIC leverages the length of the detection period to control the reporting frequency, which balances the SMux load and the detection precision.

This mechanism is lightweight in three aspects. First, the memory footprint used for tracking connection states in FPGA is minimal, with one bit per connection in the bitmap. Second, as the accessing bitmap is stored in on-chip SRAMs, the aging process will not interfere with HBM lookup in the fast path. Third, given the low signaling frequency (likely to be minutes level), the PCIe and CPU overhead are both low.

4 Implementation

We implement a fully functional prototype of Tiara with one T-switch and one T-server, equipped with one T-NIC through a PCIe Gen3 x16 link. T-switch and T-NIC are connected via 100G Ethernet cables. In the rest of this section, we discuss the implementation details of each component.

4.1 T-switch

We build a P4 prototype of T-switch with a Barefoot Tofino switch, where one pipeline has 12 physical stages, each with 1.25 MB SRAMs and 528 KB TCAMs.

[5]The aging procedure in the FCT is implemented by the SMux, which should provide a longer life cycle for a typical entry compared to the OCT due to its larger memory space.

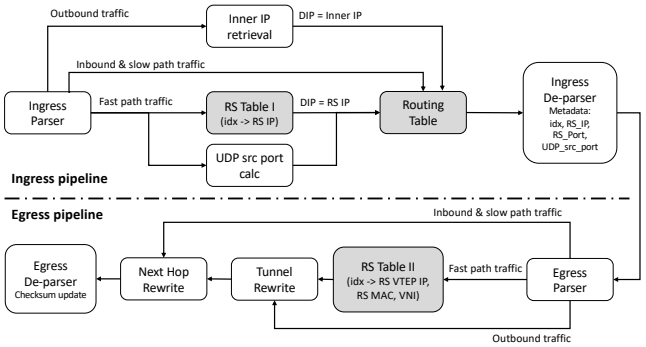

Figure 7: T-switch pipeline implementation.

ETH	Metadata				IP	...
EtherType = 0x88B5	RS_Index (4 B)	RS_Port (2 B)	RSS (2 B)	EtherType = IPv4/IPv6 (2 B)		

Figure 8: The metadata format.

We modify a baseline switch.p4[6] to implement the packet processing pipeline, including the RS table, routing tables (forwarding table and ECMP table), and tunnel processing (VxLAN encapsulation and decapsulation). Figure 7 shows an overall pipeline of T-switch. It is worth mentioning that we split the RS table into two parts. The RS table I (`RS_Index` $\rightarrow$ `RS_IP`) exists in the ingress pipeline, where T-switch retrieves `RS_IP` for routing tables lookup. T-switch postpones the lookup of the rest values in RS table II (`RS_Index` $\rightarrow$ `RS_VTEP_IP, VNI, RS_MAC`), to the egress pipeline. This decoupling helps mitigate resource contention between RS Table and routing tables in the ingress pipeline.

The modified switch.p4 takes 53.85% of SRAMs and 13.19% of TCAMs to implement the pipeline described in Figure 7 with 64K RS table entries, 2K IPv4 addresses, 1K IPv4 prefixes, 1K IPv6 addresses, and 1K IPv6 prefixes.

Recall that, in slow path processing, the VxLAN source port is computed based on the last 2 bytes of RSS value (§3.2.2). To be consistent with the slow path, the fast path in T-switch should compute this field in the same way. However, T-switch pipeline does not support the Toeplitz hash [18, 26, 30] used in RSS computation[7]. To address this issue, T-NIC carries the computed RSS value (last 2 bytes) on packets within an extended metadata header to T-switch (§4.2). T-switch retrieves the hash value from the packet and performs the same computations as SMuxes. In our implementation, the VxLAN source port is computed as followed: $port = (RSS \wedge (65535 - 49152)) + 49152$.

[6] A simplified version can be found at https://github.com/p4lang/switch

[7] We follow the standard RSS computation procedure for compatibility

4.2 T-NIC

T-NIC is implemented in a Xilinx FPGA-based SmartNIC card, with two 100GE ports and one HBM stack of 16 256MB memory channels. We use Xilinx QDMA IP [11] as the DMA engine. We implement the T-NIC logic described in Figure 4, in System Verilog, including the OCT management and lookup, packet metadata encapsulation, entry aging, and the slow path delivery.

Tiara relies on a metadata header in the packet to pass information between T-NIC and T-switch. Figure 8 shows the format of the metadata header, which is inserted between the Ethernet header and the IP header. The metadata header includes a 4-byte `RS_Index`, a 2-byte `RS_Port`, a 2-byte `RSS`, and a 2-byte `EtherType`. The field `EtherType` in the metadata header follows the IEEE 802 standard [6] to indicate the following header type (IPv4 or IPv6). In the Ethernet header, the original `EtherType` field is changed to 0x88B5, which indicates the next header is private. To avoid the drop of oversized packets caused by inserting the metadata header, we increase the MTU of T-NIC and the corresponding T-switch ports by 10 bytes, i.e., the size of the metadata header.

4.3 T-server

T-server contains 2 Intel(R) Xeon(R) Platinum 8260 CPU. We run SMuxes and offloading engines in one CPU in the same NUMA node as T-NIC without hyper-threading. We build a T-NIC driver as a DPDK [2] PMD and implement the offloading engine on top of it. We leverage an in-house SMux implementation modified from DPVS [3]. The SMux has been deployed over three years, and we make necessary changes to adapt it for the Tiara architecture. The hash computation used in both SMuxes and T-NICs is the CRC32 algorithm.

We optimize the DMA transmission between T-NIC and the PMD. QDMA is a type of Scatter-Gather DMA from Xilinx [11]. For any DMA transaction, it first reads a descriptor from the host to get the physical address of the DMA buffer. The speed of descriptor filling affects the DMA performance. Tiara leverages SIMD instructions provided by Intel processors [7] to accelerate the descriptor filling. For example, we use `_mm_storeu_si128` and `_mm_storeu_si128` to copy the DMA information between the DPDK `mbuf` and the QDMA descriptor. Moreover, Tiara decouples DMA control channels from data channels to avoid head of line blocking and mutual interference. Tiara guarantees lossless control channels by fine-grained credit control between T-server and T-NIC while remaining data channels to be lossy like conventional NIC data paths.

5 Evaluation

In this section, we use testbed experiments to evaluate the Tiara prototype as described in §4. We first show the micro-

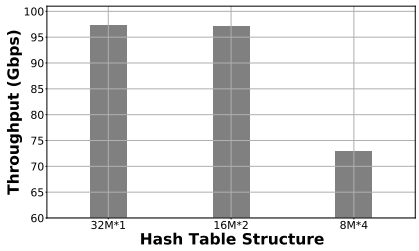

Figure 9: HBM lookup performance on different hash table structures with 10M flows.

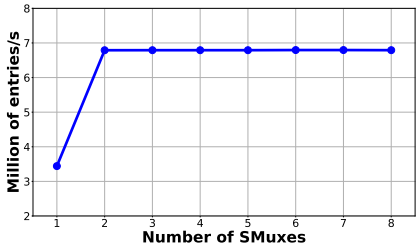

Figure 10: Entry insertion speed of a single offloading engine.

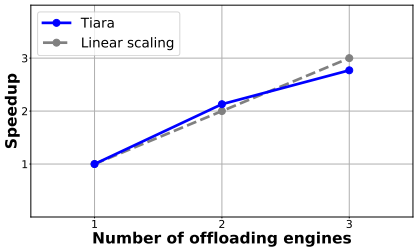

Figure 11: Insertion speedup with multiple offloading engines.

benchmarks to assess the effectiveness of Tiara component designs (§5.1). Then, we measure the end-to-end system performance of Tiara (§5.2). Last, we compare Tiara with existing approaches, i.e., SMux and Silkroad [31] (§5.3). Our results reveal that:

- A T-server with a single T-NIC can provide 200 Gbps throughput with 10M concurrent flows and could scale linearly up to 1.6 Tbps and 80M concurrent flows by hosting 8 T-NICs within a T-server.
- Tiara fast path can provide less than 4 us average latency with small jitter even at line rate.
- Tiara can serve up to 1.8M new connections per second, which is larger than switch-based HMux.
- Tiara is cost-, energy-, and space-efficient, costing 17.4 × less money, consuming 12.8 × less energy, and taking 16.8 × less rack space than SMux, given the same target throughput.

Testbed setup. We leverage the same SMux used in the Tiara slow path as the baseline of software LBs. The Tiara prototype and the baseline are directly connected to the traffic generator using 100 Gbps cables. Test traffic is generated by a hardware traffic generator, sent to the LB (Tiara or baseline), and then routed back to the generator. In this way, we could test the throughput and latency for both Tiara and the baseline.

Traffic. We use a hardware generator to inject synthetic TCP/UDP flows. Since we do not hold any assumption on traffic patterns in Tiara design, the traffic is generated in a random manner.

5.1 Micro-benchmarks

A few micro-benchmarks are designed to evaluate the major component optimizations described in §3.3. Specifically, we evaluate the lookup performance of our hash table design, measure the insertion speed of offloading engines, and test the PCIe overhead incurred by our aging mechanism.

Tiara OCT provides line-rate lookup. We run a benchmark with 10M flows in the OCT, implemented with three candidate hash table structures described in §3.3.1, i.e., 32M×1, 16M×2, and 8M×4. Figure 9 shows the lookup throughput on 10M flows with 128-byte packet size in three candidate hash structures. It reveals that both 32M×1 and 16M×2 structures approach line rate (97.2 Gbps and 97.15 Gbps), but 16M×2 provides a lower theoretic hash collision rate. When the width expands to 4, the throughput drops to 72.9 Gbps since all channels are used for each access at this width. This benchmark is consistent with our analysis in §3.3.1.

Tiara offloading engine achieves fast entry offloading. We randomly generate new flow entries in SMuxes and offload them to T-NIC by offloading engines. Therefore, in this experiment, all offloading operations are entry insertions. Figure 10 demonstrates the offloading speed of a single offloading engine, which is shared among SMuxes. The speed sticks to 6.8M operations per second with more than two SMuxes, which is bounded by the offloading engine. Figure 11 further shows how offloading speed changes with more offloading engines working in parallel. It achieves near linear-scaling with 2.77× speedup when using three offloading engines. The linear scalability of offloading speed makes Tiara able to support a high CPS scenario. For example, the LB in [5] processes 6.9M CPS, requiring at least 13.8M offloading operations (insertion or deletion), which can be supported by two offloading engines, as shown in Figure 11.

Tiara entry aging mechanism incurs negligible overhead. We evaluate the PCIe utilization caused by the aging mechanism, i.e., sending signals (packet headers) to SMuxes via PCIe, with 10M flows and a 1 minute detection period. The average PCIe utilization is less than 0.05%. Given that the control plane and data plane share the same PCIe interface, the low PCIe utilization of Tiara aging mechanism incurs little influence on the data plane.

5.2 Tiara Performance

A complete Tiara system consists of at least one T-switch connected by multiple T-servers, each hosting up to 8 200GE T-NICs. We will show in this section whether such a system could meet the design goals: > 1 Tbps, > 10M concurrent flows, and > 1M CPS, without any assumption on traffic pat-

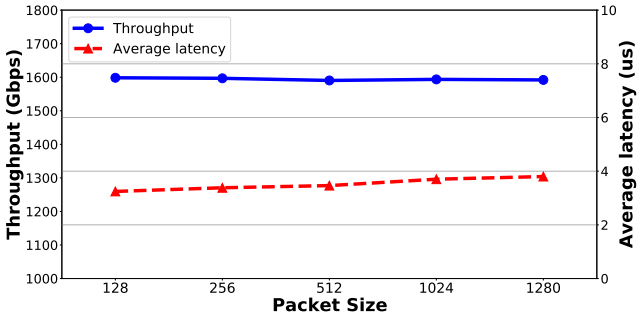

Figure 12: Forwarding performance in Tiara fast path. A T-server with 8 T-NICs achieves up to 1.6 Tbps with less than 4 us latency.

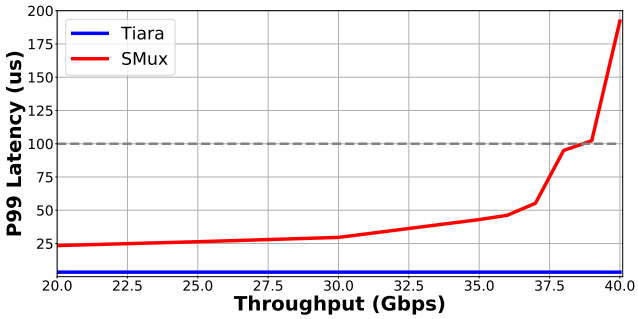

Figure 13: Latency-bounded throughput. With the tail (P99) latency bound of 100us, Tiara can achieve 200Gbps per T-NIC and 1.6Tbps per T-server with 8 T-NICs. However, since SMux suffers from high jitter when the load increases, the maximum latency-bounded throughput of SMux is 38Gbps.

terns.

Throughput and latency. To measure the throughput and latency of Tiara with 10M concurrent flows, we generate traffic consisting of 10M flows and send them to Tiara, which offloads these flows into the fast path.

We first test the performance of Tiara fast path with a single T-NIC. It can achieve the line rate of 200 Gbps and provide an extremely low average latency of less than 4 us, with packet sizes ranging from 128 to 1280 bytes. We further break down the latency distribution in Tiara fast path, which shows about 1 : 1 latency between T-switch and T-NIC.

The throughput and the number of concurrent flows supported in one T-server can scale linearly with the number of T-NICs, as T-NICs plugged in the same server are totally independent of each other, and they share nothing in the fast path processing. As a result, with 8 T-NICs in one T-server, the aggregate throughput of T-server fast path scales linearly to 1.6 Tbps, and the latency remains exactly the same as that of a single T-NIC (i.e., less than 4 us), as shown in Figure 12. Similarly, the number of concurrent flows increases to 80M for a holistic T-server with 8 T-NICs. If the throughput requirement of an LB system is larger than 1.6 Tbps or the flow number requirement is larger than 80M, more T-servers can be connected to the T-switch tier, given the flexibility of this architecture. The aggregate throughput and the flow capacity of Tiara in the fast path can also scale linearly with the number of T-servers, as they are physically independent as well.

CPS. We evaluate Tiara ability to serve new TCP connections by issuing HTTP transactions, including a TCP connection establishment, an HTTP GET request, an HTTP response (bypassing LB), and a closure of TCP connection. We gradually increase the target CPS in 0.1M granularity at the generator to find the maximum available CPS that the target LB can serve all the incoming requests. The result reveals that Tiara can support up to 1.8M CPS (bounded by SMux), which is higher than our goal (i.e., 1M CPS).

Resilience to traffic patterns. By leveraging the large capacity of FPGA HBM for the connection table, almost all flows can be offloaded to the fast path in Tiara as long as the number of concurrent flows is less than 10M per T-NIC and 80M per T-server, which is true in most cases as we observed at our datacenter boundaries. As a result, Tiara is insensitive to traffic patterns, and it keeps consistent high throughput and low latency on different traffic patterns.

5.3 Tiara vs. Existing Approaches

In this section, we compare Tiara with existing approaches (the SMux baseline, Silkroad [31]) in terms of performance and efficiency. The results are summarized in Table 1.

Performance. SMux suffers from high latency and jitter when the traffic load is heavy [33] due to high CPU utilization and cache misses. High latency and jitter will adversely impact the user's network experience. Therefore, we use "latency-bounded throughput" as the metric to compare SMux and Tiara more fairly. Given that the end-to-end latency from Internet users to datacenter services could be as low as a few milliseconds [35, 39], we should bound the tail latency of LB to the sub-millisecond level to minimize its impact on the user's network experience. In this experiment, we run SMux on 16 cores of a server with the same configuration as T-server (§4.3), except that the SMux server is equipped with a 100 Gbps Mellanox ConnectX-5 NIC rather than T-NICs. We set the bound of LB P99 latency to 100 us and compare the latency-bounded throughput with the packet size of 512 bytes between Tiara and SMux. Figure 13 shows the P99 latency of Tiara and SMux, respectively, with different throughputs. For the Tiara fast path, P99 latency is consistently below 4 us at throughput up to 200 Gbps per T-NIC, while SMux P99 latency breaks the 100 us bound when the throughput is higher than 38 Gbps. Therefore, we consider 38 Gbps as the maximum latency-bounded throughput of the baseline SMux. In Tiara, the latency-bounded throughput of a single T-server with 8 T-NICs is 1.6 Tbps, 42.1× higher than SMux (38 Gbps), and its P99 latency (4 us) is 25× lower than

	Throughput	P99 lat.	CPS	CT size*	Cost efficiency	Energy efficiency	Space efficiency
SMux	38 Gbps	100 us	1.8M	~100 GB	4.75 Gbps/(cost unit)	76 Mbps/Watt	19 Gbps/U
Silkroad**	1.6 Tbps	< 2 us	200K	100 MB	457.14 Gbps/(cost unit)	2909.1 Mbps/Watt	1600 Gbps/U
Tiara	1.6 Tbps	< 4 us	1.8M	4 GB	82.05 Gbps/(cost unit)	969.7 Mbps/Watt	320 Gbps/U

Table 1: Performance and efficiency comparison among different LBs. *Since the connection table (CT) compression in Silkroad is orthogonal to Tiara and can be applied in any architecture, we use the CT size as the metric to compare the data plane scalability of different architectures. **The Silkroad paper does not report throughput and tail latency explicitly, and we use the same throughput and latency results as T-switch to simplify comparison.

SMux (100 us).

Silkroad achieves comparable high throughput and low latency as Tiara, as most connections are processed in the hardware fast path in both solutions. However, Silkroad is less scalable in both control and data paths than Tiara. Silkroad leverages the embedded management CPU in switch for connection creation and offloading, thus expecting only 200K CPS [31]. Tiara achieves 1.8M CPS, 9 × higher than Silkroad, thanks to the optimizations in the control plane of Tiara. Silkroad stores the connection table in the switch's limited on-chip SRAMs. Despite compression with hash digest, the connection table is still bounded by the on-chip SRAM size, i.e., 50-100 MB in modern switching ASICs. Tiara leverages 4 GB HBM in modern FPGA, increasing the connection table size in the fast path by orders of magnitude compared to Silkroad.

Efficiency. In this section, we quantify and compare the efficiency of SMux, Silkroad, and Tiara, in terms of cost efficiency (performance per dollar), energy efficiency (performance per watt), and space efficiency (performance per rack unit).

- **Cost efficiency.** As the concrete cost numbers of T-NIC, T-switch, and T-server used in the Tiara prototype are confidential, we normalize them to 1, 3.5, and 8, respectively. With these cost units, the normalized system costs of SMux, Silkroad, and Tiara are 8, 3.5, and 19.5 (=3.5+1*8+8), respectively. Given these normalized system costs and the throughput data shown in Table 1, the cost efficiency of these three approaches will be 4.75 Gbps/(cost unit), 457.14 Gbps/(cost unit), and 82.05 Gbps/(cost unit), respectively.
- **Energy efficiency.** According to hardware datasheets, T-NIC, T-switch, and T-server used in the Tiara prototype consume 75 Watt, 550 Watt, and 500 Watt power, respectively. Based on these power consumption and throughput data, the energy efficiency of SMux, Silkroad, and Tiara will be 76 Mbps/Watt, 2909.1 Mbps/Watt, and 969.7 Mbps/Watt, respectively.
- **Space efficiency.** The server used in SMux is 2 rack-unit (i.e., 2U) high, the switch used in Silkroad is 1U high, and the entire Tiara system is 5U high, as it includes a 1U T-switch and a 4U T-server hosting 8 T-NICs. Based on these heights and throughput data, the space efficiency of SMux, Silkroad, and Tiara will be 19 Gbps/U, 1600 Gbps/U, and 320 Gbps/U, respectively.
- **Tiara vs. SMux in efficiency.** The cost, energy, and space efficiency of Tiara are 17.4 × , 12.8 × , and 16.8 × higher than those of SMux, respectively. In other words, given the same target throughput, Tiara costs 17.4 × less money, consumes 12.8 × less energy, and takes 16.8 × less rack space than SMux. All these efficiency advantages of Tiara over SMux come from hardware acceleration, as suitable hardware (i.e., FPGA and programmable switch in Tiara) is fundamentally much more efficient than x86 servers in network packet processing.
- **Tiara vs. Silkroad in efficiency.** As we can see from Table 1, the switch-only solution in Silkroad outperforms Tiara in all efficiency metrics. This is expected as Silkroad only leverages a switch, which is fundamentally more cost-, energy- and space-efficient than FPGA and x86 in network packet processing. However, as we discussed in the above section, the efficiency of Silkroad comes at the cost of lower CPS and smaller connection tables due to switch inherent scalability limitations. Compared to Silkroad, Tiara strikes a better balance between efficiency and scalability. Furthermore, the switch-only solution may not be that practical in traffic scenarios with a large number of connections, where Silkroad suggests operators combine its switch with an SMux for the slow path [31]. With this hybrid setting (switch + server), the efficiency of Silkroad will become similar to Tiara, but its scalability in hardware is still lower than Tiara.

One more option to further improve the efficiency of Tiara is to bake its implementation into a custom ASIC, which makes it as efficient as the Silkroad switch-only solution and as scalable as current Tiara. However, a custom ASIC incurs a significant NRE (non-recurring engineering) cost. Without a big enough volume to amortize the NRE, the cost efficiency of custom ASIC is worse than that of current Tiara design. As the performance of a single T-server is already high enough (up to 1.6 Tbps), we do not necessarily need a large number of T-servers to load-balance Internet traffic in even hyper-scale datacenters. Therefore, the design choice of using FPGA rather than custom ASIC in Tiara is justified in this context.

6 Related Work

Memory enhanced switches: eXtra Large Table (XLT) [17] enhances programmable switches with FPGA + DRAM complexes to support large tables. It works well when all rule/flow tables are stored in DRAM, and the switch and FPGA can handle all data plane processing entirely. However, that is not the case for stateful load balancers discussed in this paper. Despite large DRAM, packet lookup may still miss in XLT FPGA due to hash collision or first packet processing for new connections, but how to handle these exceptions is unclear.

TEA [25] extends switching ASIC memory virtually by utilizing the host DRAM via RDMA. However, looking up a table at the remote memory prevents switches from line-rate processing. TEA relies heavily on traffic locality that caches hot traffic in the on-chip SRAMs to preserve high throughput. Otherwise, its performance approaches the server-based lookup table, as demonstrated in its experiment (TEA with and without cache). Moreover, TEA shares the same scalability issue on the control plane as other programmable switches.

Layer-4 load balancing: There have been continuous efforts on layer-4 load balancing. In general, two LB categories are explored: stateful LBs that keep the per-connection state at Muxes and stateless LBs that do not maintain any per-connection state.

Ananta [36] and Maglev [21] are two proposed software stateful LBs with a series of packet processing optimizations, including batch processing, poll mode NIC driver, and zero-copy operations. Despite these optimizations, the packet forwarding throughput on a single server is still limited, so that they need a large number of servers to support terabits per second traffic.

Duet [24] and Rubik [23] accelerate Ananta with commodity switches in a stateless style. They store the VIP-to-DIP (`RS_IP`) mapping in switch on-chip SRAMs as an ECMP table. To support large-scale mapping rules, they leverage the tail distribution in VIP traffic to configure the heavy-hitting rules on switches while processing the rest in the software.

Beamer [33] is a recently proposed stateless LB. It relies on hash functions on the switch to proceed fast real server selection and uses "daisy chaining" techniques to mitigate the PCC violations. The "daisy chaining" requires real servers to redirect unexpected packets. However, it is empirically impractical to modify the service servers. Moreover, stateless LBs can only provide suboptimal workload balancing due to the nature of hash functions as described in [16].

Silkroad [31] is the most related work, which accelerates stateful LB with programmable switches. It faces the same problems as mentioned in §2.3, but it only focuses on addressing the data plane scalability issue with on-chip SRAMs. Silkroad stores a hash digest of a connection instead of the 5-tuple in the connection table, which reduces the key size of each connection from dozens of bytes to 16 bits. Such compression technique scales to support millions of concurrent flows. However, Silkroad will suffer from throughput degradation due to pipeline folding for those switches that distribute their SRAM resources in multiple pipelines.

Cheetah [16] aims to design a high-speed LB for both stateless and stateful manners. One of its contributions is to solve the entry insertion inefficiency problem in stateful LB by storing unused hash indexes in a connection stack. For every new coming connection, Cheetah pops an index from the connection stack and inserts the connection entry into the hash table with the retrieved index. This index, encoded in the packet header as a cookie, is carried by the connection in the following packets. The change on the packet header requires modifications on services' client sides. This requirement prevents Cheetah from deploying on large-scale datacenters with hundreds and thousands of services.

Component design & optimization: Some techniques used in the component design and optimization in Tiara have been extensively studied. Tong et al. [41] propose a high-throughput hash table structure with the idea of fixed-length hashing in FPGA DRAM. Mogul et al. [32] eliminate the livelock by a polling-based mechanism, and Kuperman et al. [27] match each net device TX queue to a hardware send queue to avoid spin-lock contention. Ross [38] splits tables into different cores in a multi-core database system to reduce the synchronization cost. The SmartNIC used in Azure [22] periodically reports flow states to the software, which allows the software manager to age the inactive flows. Our contribution is to integrate those techniques to achieve the design goals of Tiara.

7 Conclusion

Tiara is a novel hardware acceleration architecture for stateful load balancers. It simultaneously provides high throughput, low latency, high scalability, and high efficiency by mapping different LB tasks into their most suitable hardware and carefully designing and optimizing a few key components. Although we only show Tiara's capabilities to accelerate stateful load balancers in this paper, we believe this architecture is generic for network function acceleration and can be explored in the future in more gateway scenarios, such as DDoS protection and firewall.

Acknowledgments

We thank our anonymous reviewers and shepherd Anuj Kalia for their insightful comments. We also thank Naiqian Zheng, Kaicheng Yang, and Yuxuan Gao for their support of the project. The work of Chaoliang Zeng, Zilong Wang, and Kai Chen was supported in part by a ByteDance Research Collaboration Project and the Hong Kong RGC TRS T41-603/20-R and GRF 16215119.

References

[1] Axi high bandwidth memory controller v1.0. https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.

[2] Dpdk. https://www.dpdk.org/.

[3] Dpvs is a high performance layer-4 load balancer based on dpdk. https://github.com/iqiyi/dpvs.

[4] express data path. https://www.iovisor.org/technology/xdp.

[5] High-performance dpdk-based server load balancing for alibaba singles' day shopping festival. https://www.alibabacloud.com/blog/593984?spm=a2c5t.11065265.1996646101.searchclickresult.289b2f0591lA1a.

[6] Ieee 802 numbers. https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml.

[7] Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide.

[8] Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran.

[9] Load balancing 101: Nuts and bolts. https://www.f5.com/services/resources/glossary/load-balancer.

[10] Ovs conntrack. https://docs.openvswitch.org/en/latest/tutorials/ovs-conntrack/.

[11] Qdma subsystem for pci express. https://www.xilinx.com/products/intellectual-property/pcie-qdma.html.

[12] Unveiling the networks behind the 2018 double 11 global shopping festival. https://www.alibabacloud.com/blog/594167?spm=a2c5t.11065265.1996646101.searchclickresult.289b2f0575gg5Z.

[13] Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. https://tools.ietf.org/html/rfc7348.

[14] Anurag Agrawal and Changhoon Kim. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *HCS 2020*.

[15] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. Balancing on the edge: Transport affinity without network state. In *NSDI 2018*.

[16] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *NSDI 2020*.

[17] Curt Beckmann, Ramkumar Krishnamoorthy, Han Wang, Andre Lam, and Changhoon Kim. Hurdles for a dram-based match-action table. In *ICIN 2020*.

[18] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. Umac: Fast and secure message authentication. In *CRYPTO 1999*.

[19] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM 2012*.

[20] Alan Edelman. Akamai technologies: A mathematical success story. In *SIAM News 1999*.

[21] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *NSDI 2016*.

[22] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *NSDI 2018*.

[23] Rohan Gandhi, Y Charlie Hu, Cheng-Kok Koh, Hongqiang Harry Liu, and Ming Zhang. Rubik: unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *ATC 2015*.

[24] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM 2014*.

[25] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *SIGCOMM 2020*.

[26] Hugo Krawczyk. Lfsr-based hashing and authentication. In *CRYPTO 1994*.

[27] Yossi Kuperman, Maxim Mikityanskiy, and Rony Efraim. Hierarchical qos hardware offload (htb).

[28] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *SOSP 2017*.

[29] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *SIGCOMM 2016*.

[30] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. In *TCS 1993*.

[31] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *SIGCOMM 2017*.

[32] Jeffrey C Mogul and KK Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *TOCS 1997*.

[33] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *NSDI 2018*.

[34] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Journal of Algorithms 2004*.

[35] Fabio Palumbo, Giuseppe Aceto, Alessio Botta, Domenico Ciuonzo, Valerio Persico, and Antonio Pescapé. Characterization and analysis of cloud-to-user latency: the case of azure and aws. In *CN 2021*.

[36] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *SIGCOMM 2013*.

[37] Kun Qian, Sai Ma, Mao Miao, Jianyuan Lu, Tong Zhang, Peilong Wang, Chenghao Sun, and Fengyuan Ren. Flexgate: High-performance heterogeneous gateway in data centers. In *APNet 2019*.

[38] Kenneth A Ross. Multicore processors and database systems: The multicore transformation. In *Ubiquity 2014*.

[39] Ao-Jan Su, David R Choffnes, Aleksandar Kuzmanovic, and Fabian E Bustamante. Drafting behind akamai: Inferring network conditions based on cdn redirections. In *TON 2009*.

[40] Ao-Jan Su and Aleksandar Kuzmanovic. Thinning akamai. In *SIGCOMM 2008*.

[41] Da Tong, Shijie Zhou, and Viktor K Prasanna. High-throughput online hash table on fpga. In *IPDPS 2015*.

[42] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. Fpga-accelerated compactions for lsm-based key-value store. In *FAST 2020*.

A Analysis on Hash Collision

Suppose there are n random entries inserted into a hash table with width w and depth d. The probability that any i entries are hashed to the same index is:

$$p(i) = C_n^i (\frac{1}{d})^i (1-\frac{1}{d})^{n-i} \tag{1}$$

The probability for any indexes that hold $0 \sim w$ entries is:

$$p(num \leq w) = \sum_{i=0}^{w} C_n^i (\frac{1}{d})^i (1-\frac{1}{d})^{n-i} \tag{2}$$

For all indexes, this probability becomes:

$$p(num \leq w)_{all} = (\sum_{i=0}^{w} C_n^i (\frac{1}{d})^i (1-\frac{1}{d})^{n-i})^d \tag{3}$$

Therefore, the probability for all indexes that exist at least once collision, i.e., holding more than w entries, is:

$$p(num > w)_{all} = 1 - (\sum_{i=0}^{l} C_n^i (\frac{1}{d})^i (1-\frac{1}{d})^{n-i})^d \tag{4}$$

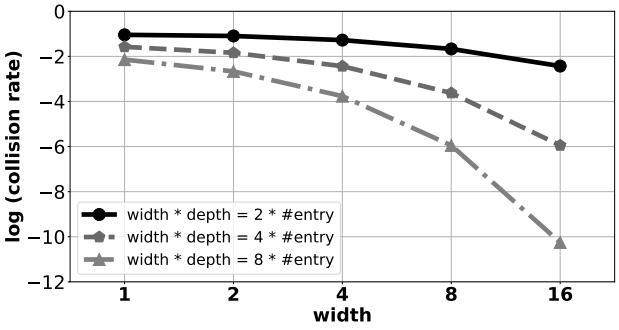

Figure 14: The numerical simulation on collision rates in different widths and depths with #entry = 32768. The collision rates are shown in the log scale.

To get an intuitive relationship between the collision rate and the width, we conduct a numerical simulation on different settings based on Equation 4. The results are demonstrated in Figure 14, and show that given a fixed hash space ($> n$), a larger width results in a lower hash collision rate.