Maglev: A Fast and Reliable Software Network Load Balancer

Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith,
Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney,
Wentao Shang[†*] and Jinnah Dylan Hosein[‡*]

Google Inc. [†]UCLA [‡]SpaceX
maglev-nsdi@google.com

Abstract

Maglev is Google's network load balancer. It is a large distributed software system that runs on commodity Linux servers. Unlike traditional hardware network load balancers, it does not require a specialized physical rack deployment, and its capacity can be easily adjusted by adding or removing servers. Network routers distribute packets evenly to the Maglev machines via Equal Cost Multipath (ECMP); each Maglev machine then matches the packets to their corresponding services and spreads them evenly to the service endpoints. To accommodate high and ever-increasing traffic, Maglev is specifically optimized for packet processing performance. A single Maglev machine is able to saturate a 10Gbps link with small packets. Maglev is also equipped with consistent hashing and connection tracking features, to minimize the negative impact of unexpected faults and failures on connection-oriented protocols. Maglev has been serving Google's traffic since 2008. It has sustained the rapid global growth of Google services, and it also provides network load balancing for Google Cloud Platform.

1 Introduction

Google is a major source of global Internet traffic [29, 30]. It provides hundreds of user-facing services, in addition to many more services hosted on the rapidly growing *Cloud Platform* [6]. Popular Google services such as *Google Search* and *Gmail* receive millions of queries per second from around the globe, putting tremendous demand on the underlying serving infrastructure.

To meet such high demand at low latency, a Google service is hosted on a number of servers located in multiple clusters around the world. Within each cluster, it is essential to distribute traffic load evenly across these servers in order to utilize resources efficiently so that no single server gets overloaded. As a result, network load balancers form a critical component of Google's production network infrastructure.

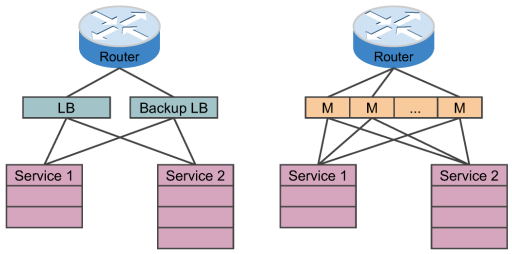

Figure 1: Hardware load balancer and Maglev.

A network load balancer is typically composed of multiple devices logically located between routers and service endpoints (generally TCP or UDP servers), as shown in Figure 1. The load balancer is responsible for matching each packet to its corresponding service and forwarding it to one of that service's endpoints.

Network load balancers have traditionally been implemented as dedicated hardware devices [1, 2, 3, 5, 9, 12, 13], an approach that has several limitations. First, their scalability is generally constrained by the maximum capacity of a single unit, making it impossible to keep up with Google's traffic growth. Second, they do not meet Google's requirements for high availability. Though often deployed in pairs to avoid single points of failure, they only provide *1+1 redundancy*. Third, they lack the flexibility and programmability needed for quick iteration, as it is usually difficult, if not impossible, to modify a hardware load balancer. Fourth, they are costly to upgrade. Augmenting the capacity of a hardware load balancer usually involves purchasing new hardware as well as physically deploying it. Because of all these limitations, we investigated and pursued alternative solutions.

With all services hosted in clusters full of commodity servers, we can instead build the network load balancer as a distributed software system running on these servers. A software load balancing system has many advantages

[*]Work was done while at Google.

over its hardware counterpart. We can address scalability by adopting the *scale-out* model, where the capacity of the load balancer can be improved by increasing the number of machines in the system: through ECMP forwarding, traffic can be evenly distributed across all machines. Availability and reliability are enhanced as the system provides *N+1 redundancy*. By controlling the entire system ourselves, we can quickly add, test, and deploy new features. Meanwhile, deployment of the load balancers themselves is greatly simplified: the system uses only existing servers inside the clusters. We can also divide services between multiple *shards* of load balancers in the same cluster in order to achieve performance isolation.

Despite all the benefits, the design and implementation of a software network load balancer are highly complex and challenging. First, each individual machine in the system must provide high throughput. Let N be the number of machines in the system and T be the maximum throughput of a single machine. The maximum capacity of the system is bounded by $N \times T$. If T is not high enough, it will be uneconomical for the system to provide enough capacity for all services [22]. The system as a whole must also provide *connection persistence*: packets belonging to the same connection should always be directed to the same service endpoint. This ensures quality of service as clusters are very dynamic and failures are quite common [23, 40].

This paper presents Maglev, a fast and reliable software network load balancing system. Maglev has been a critical component of Google's frontend serving infrastructure since 2008, and currently serves almost all of Google's incoming user traffic. By exploiting recent advances in high-speed server networking techniques [18, 41, 35, 31], each Maglev machine is able to achieve line-rate throughput with small packets. Through *consistent hashing* and *connection tracking*, Maglev provides reliable packet delivery despite frequent changes and unexpected failures. While some of the techniques described in this paper have existed for years, this paper shows how to build an operational system using these techniques. The major contributions of this paper are to: 1) present the design and implementation of Maglev, 2) share experiences of operating Maglev at a global scale, and 3) demonstrate the capability of Maglev through extensive evaluations.

2 System Overview

This section provides an overview of how Maglev works as a network load balancer. We give a brief introduction to Google's frontend serving architecture, followed by a description of how the Maglev system is configured.

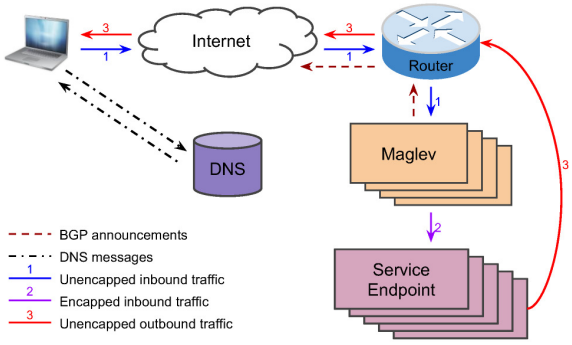

Figure 2: Maglev packet flow.

2.1 Frontend Serving Architecture

Maglev is deployed in Google's frontend-serving locations, including clusters of varying sizes. For simplicity, we only focus on the setup in the smaller clusters in this paper, and briefly describe the larger cluster setup below. Figure 2 shows an overview of Google's frontend serving architecture in the small cluster setup.

Every Google service has one or more *Virtual IP addresses* (VIPs). A VIP is different from a physical IP in that it is not assigned to a specific network interface, but rather served by multiple service endpoints behind Maglev. Maglev associates each VIP with a set of service endpoints and announces it to the router over BGP; the router in turn announces the VIP to Google's backbone. Aggregations of the VIP networks are announced to the Internet to make them globally accessible. Maglev handles both IPv4 and IPv6 traffic, and all the discussion below applies equally to both.

When a user tries to access a Google service served on *www.google.com*, her browser first issues a DNS query, which gets a response (possibly cached) from one of Google's authoritative DNS servers. The DNS server assigns the user to a nearby frontend location taking into account both her geolocation and the current load at each location, and returns a VIP belonging to the selected location in response [16]. The browser will then try to establish a new connection with the VIP.

When the router receives a VIP packet, it forwards the packet to one of the Maglev machines in the cluster through ECMP, since all Maglev machines announce the VIP with the same cost. When the Maglev machine receives the packet, it selects an endpoint from the set of service endpoints associated with the VIP, and encapsulates the packet using *Generic Routing Encapsulation* (GRE) with the outer IP header destined to the endpoint.

When the packet arrives at the selected service endpoint, it is decapsulated and consumed. The response, when ready, is put into an IP packet with the source address being the VIP and the destination address being

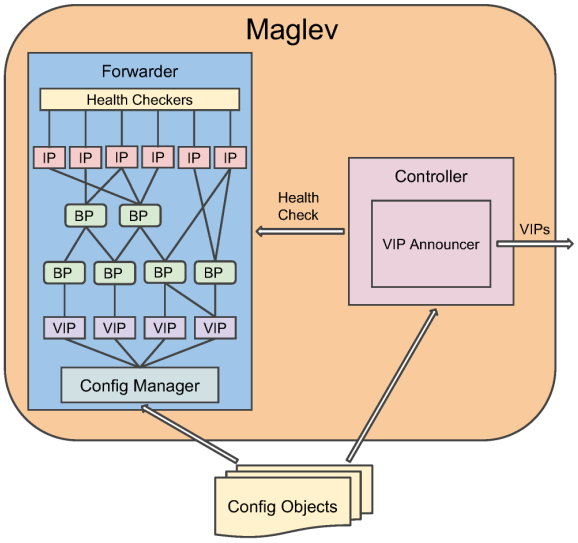

Figure 3: Maglev config (BP stands for backend pool).

the IP of the user. We use *Direct Server Return* (DSR) to send responses directly to the router so that Maglev does not need to handle returning packets, which are typically larger in size. This paper focuses on the load balancing of incoming user traffic. The implementation of DSR is out of the scope of this paper.

The setup for large clusters is more complicated: to build clusters at scale, we want to avoid the need to place Maglev machines in the same layer-2 domain as the router, so hardware encapsulators are deployed behind the router, which tunnel packets from routers to Maglev machines.

2.2 Maglev Configuration

As described in the previous subsection, Maglev is responsible for announcing VIPs to the router and forwarding VIP traffic to the service endpoints. Therefore, each Maglev machine contains a controller and a forwarder as depicted in Figure 3. Both the controller and the forwarder learn the VIPs to be served from configuration objects, which are either read from files or received from external systems through RPC.

On each Maglev machine, the controller periodically checks the health status of the forwarder. Depending on the results, the controller decides whether to announce or withdraw all the VIPs via BGP. This ensures the router only forwards packets to healthy Maglev machines.

All VIP packets received by a Maglev machine are handled by the forwarder. At the forwarder, each VIP is configured with one or more backend pools. Unless otherwise specified, the backends for Maglev are service endpoints. A backend pool may contain the physical IP addresses of the service endpoints; it may also recursively contain other backend pools, so that a frequently-used set of backends does not need to be specified repeatedly. Each backend pool, depending on its specific requirements, is associated with one or more health checking methods with which all its backends are verified; packets will only be forwarded to the healthy backends. As the same server may be included in more than one backend pool, health checks are deduplicated by IP addresses to avoid extra overhead.

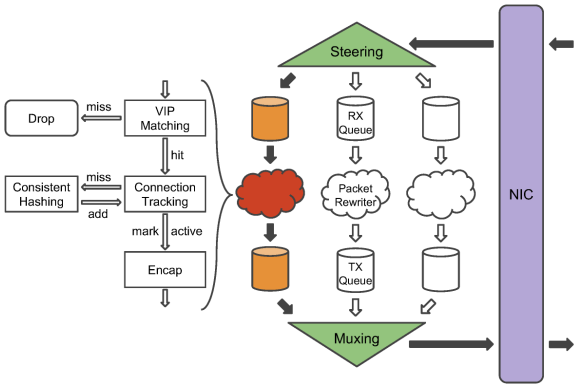

Figure 4: Maglev forwarder structure.

The forwarder's *config manager* is responsible for parsing and validating config objects before altering the forwarding behavior. All config updates are committed atomically. Configuration of Maglev machines within the same cluster may become temporarily out of sync due to delays in config push or health checks. However, consistent hashing will make connection flaps between Maglevs with similar backend pools mostly succeed even during these very short windows.

It is possible to deploy multiple *shards* of Maglevs in the same cluster. Different Maglev shards are configured differently and serve different sets of VIPs. Sharding is useful for providing performance isolation and ensuring quality of service. It is also good for testing new features without interfering with regular traffic. For simplicity, we assume one shard per cluster in this paper.

3 Forwarder Design and Implementation

The forwarder is a critical component of Maglev, as it needs to handle a huge number of packets quickly and reliably. This section explains the design and implementation details of the key modules of the Maglev forwarder, as well as the rationale behind the design.

3.1 Overall Structure

Figure 4 illustrates the overall structure of the Maglev forwarder. The forwarder receives packets from the

NIC (Network Interface Card), rewrites them with proper GRE/IP headers and then sends them back to the NIC. The Linux kernel is not involved in this process.

Packets received by the NIC are first processed by the *steering module* of the forwarder, which calculates the *5-tuple hash*[1] of the packets and assigns them to different *receiving queues* depending on the hash value. Each receiving queue is attached to a packet rewriter thread. The packet thread first tries to match each packet to a configured VIP. This step filters out unwanted packets not targeting any VIP. Then it recomputes the 5-tuple hash of the packet and looks up the hash value in the *connection tracking table* (covered in Section 3.3). We do not reuse the hash value from the steering module to avoid cross-thread synchronization.

The connection table stores backend selection results for recent connections. If a match is found and the selected backend is still healthy, the result is simply reused. Otherwise the thread consults the *consistent hashing module* (covered in Section 3.4) and selects a new backend for the packet; it also adds an entry to the connection table for future packets with the same 5-tuple. A packet is dropped if no backend is available. The forwarder maintains one connection table per packet thread to avoid access contention. After a backend is selected, the packet thread encapsulates the packet with proper GRE/IP headers and sends it to the attached *transmission queue*. The *muxing module* then polls all transmission queues and passes the packets to the NIC.

The steering module performs 5-tuple hashing instead of round-robin scheduling for two reasons. First, it helps lower the probability of packet reordering within a connection caused by varying processing speed of different packet threads. Second, with connection tracking, the forwarder only needs to perform backend selection once for each connection, saving clock cycles and eliminating the possibility of differing backend selection results caused by race conditions with backend health updates. In the rare cases where a given receiving queue fills up, the steering module falls back to round-robin scheduling and spreads packets to other available queues. This fallback mechanism is especially effective at handling large floods of packets with the same 5-tuple.

3.2 Fast Packet Processing

The Maglev forwarder needs to process packets as fast as possible in order to cost-effectively scale the serving capacity to the demands of Google's traffic. We engineered it to forward packets at line rate – typically 10Gbps in Google's clusters today. This translates to 813Kpps (packets per second) for 1500-byte IP packets.

[1]The 5-tuple of a packet refers to the source IP, source port, destination IP, destination port and IP protocol number.

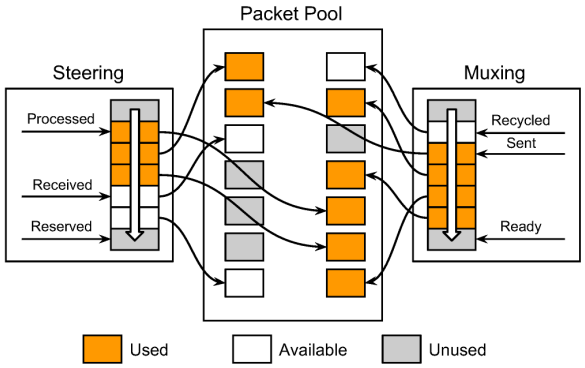

Figure 5: Packet movement into and out of the forwarder.

However, our requirements are much more stringent: we must handle very small packets effectively because incoming requests are typically small in size. Assuming IP packet size is 100 bytes on average, the forwarder must be able to process packets at 9.06Mpps. This subsection describes the key techniques we employed to reach and exceed this packet processing speed.

Maglev is a userspace application running on commodity Linux servers. Since the Linux kernel network stack is rather computationally expensive, and Maglev doesn't require any of the Linux stack's features, it is desirable to make Maglev bypass the kernel entirely for packet processing. With proper support from the NIC hardware, we have developed a mechanism to move packets between the forwarder and the NIC without any involvement of the kernel, as shown in Figure 5. When Maglev is started, it pre-allocates a packet pool that is shared between the NIC and the forwarder. Both the steering and muxing modules maintain a *ring queue* of pointers pointing to packets in the packet pool.

Both the steering and muxing modules maintain three pointers to the rings. At the receiving side, the NIC places newly received packets at the *received* pointer and advances it. The steering module distributes the received packets to packet threads and advances the *processed* pointer. It also reserves unused packets from the packet pool, places them into the ring and advances the *reserved* pointer. The three pointers chase one another as shown by the arrows. Similarly, on the sending side the NIC sends packets pointed to by the *sent* pointer and advances it. The muxing module places packets rewritten by packet threads into the ring and advances the *ready* pointer. It also returns packets already sent by the NIC back to the packet pool and advances the *recycled* pointer. Note that the packets are not copied anywhere by the forwarder.

To reduce the number of expensive boundary-crossing operations, we process packets in batches whenever possible. In addition, the packet threads do not share any

data with each other, preventing contention between them. We pin each packet thread to a dedicated CPU core to ensure best performance. With all these optimizations, Maglev is able to achieve line rate with small packets, as shown in Section 5.2.

Further, the latency that Maglev adds to the path taken by each packet is small. Normally it takes the packet thread about 350ns to process each packet on our standard servers. There are two special cases in which packet processing may take longer. Since the forwarder processes packets in batches, each batch is processed when it grows large enough or when a periodic timer expires. In practice we set the timer to be 50μs. Therefore if Maglev is significantly underloaded, a 50μs delay will be added to each packet in the worst case. One possible optimization to this case is to adjust batch sizes dynamically [32]. The other case where Maglev may add extra processing delay is when Maglev is overloaded. The maximum number of packets that Maglev can buffer is the size of the packet pool; beyond that the packets will be dropped by the NIC. Assuming the packet pool size is 3000 and the forwarder can process 10Mpps, it takes about 300μs to process all buffered packets. Hence a maximum of 300μs delay may be added to each packet if Maglev is heavily overloaded. Fortunately, this case can be avoided by proper capacity planning and adding Maglev machines as needed.

3.3 Backend Selection

Once a packet is matched to a VIP, we need to choose a backend for the packet from the VIP's backend pool. For connection-oriented protocols such as TCP, it is critical to send all packets of a connection to the same backend. We accomplish this with a two part strategy. First, we select a backend using a new form of consistent hashing which distributes traffic very evenly. Then we record the selection in a local connection tracking table.

Maglev's connection tracking table uses a fixed-size hash table mapping 5-tuple hash values of packets to backends. If the hash value of a packet does not exist in the table, Maglev will assign a backend to the packet and store the assignment in the table. Otherwise Maglev will simply reuse the previously assigned backend. This guarantees that packets belonging to the same connection are always sent to the same backend, as long as the backend is still able to serve them. Connection tracking comes in handy when the set of backends changes: for instance, when backends go up and down, are added or removed, or when the backend weights change.

However, per-Maglev connection tracking alone is insufficient in our distributed environment. First, it assumes all packets with the same 5-tuple are always sent to the same Maglev machine. Because the router in front of Maglev does not usually provide connection affinity, this assumption does not hold when the set of Maglev machines changes. Unfortunately, such changes are inevitable and may happen for various reasons. For example, when upgrading Maglevs in a cluster we do a rolling restart of machines, draining traffic from each one a few moments beforehand and restoring it once the Maglev starts serving again. This process may last over an hour, during which the set of Maglevs keeps changing. We also sometimes add, remove, or replace Maglev machines. All of these operations make standard ECMP implementations shuffle traffic on a large scale, leading to connections switching to different Maglevs in mid-stream. The new Maglevs will not have the correct connection table entries, so if backend changes occur at the same time, connections will break.

A second theoretical limitation is that the connection tracking table has finite space. The table may fill up under heavy load or SYN flood attacks. Since Maglev only evicts entries from the connection table when they are expired, once the table becomes full, we will need to select a backend for each packet that doesn't fit in the table. While in practice there is plenty of memory on a modern machine, in deployments where we share machines between Maglev and other services, we may need to sharply limit the connection table size.

If any of the above cases occur, we can no longer rely on connection tracking to handle backend changes. Thus Maglev also provides consistent hashing to ensure reliable packet delivery under such circumstances.

3.4 Consistent Hashing

One possible approach to address the limitations of connection tracking is to share connection state among all Maglev machines, for example in a distributed hash table as suggested in [34]. However, this would negatively affect forwarding performance – recall that connection states are not even shared among packet threads on the same Maglev machine to avoid contention.

A better-performing solution is to use local consistent hashing. The concept of consistent hashing [28] or rendezvous hashing [38] was first introduced in the 1990s. The idea is to generate a large lookup table with each backend taking a number of entries in the table. These methods provide two desirable properties that Maglev also needs for resilient backend selection:

- *load balancing*: each backend will receive an almost equal number of connections.
- *minimal disruption*: when the set of backends changes, a connection will likely be sent to the same backend as it was before.

Pseudocode 1 Populate Maglev hashing lookup table.

```
1:  function POPULATE
2:      for each i < N do next[i] ← 0 end for
3:      for each j < M do entry[j] ← −1 end for
4:      n ← 0
5:      while true do
6:          for each i < N do
7:              c ← permutation[i][next[i]]
8:              while entry[c] ≥ 0 do
9:                  next[i] ← next[i] + 1
10:                 c ← permutation[i][next[i]]
11:             end while
12:             entry[c] ← i
13:             next[i] ← next[i] + 1
14:             n ← n + 1
15:             if n = M then return end if
16:         end for
17:     end while
18: end function
```

Both [28] and [38] prioritize minimal disruption over load balancing, as they were designed to optimize web caching on a small number of servers. However, Maglev takes the opposite approach for two reasons. First, it is critical for Maglev to balance load as evenly as possible among the backends. Otherwise the backends must be aggressively overprovisioned in order to accommodate the peak traffic. Maglev may have hundreds of backends for certain VIPs, our experiments show that both [28] and [38] will require a prohibitively large lookup table for each VIP to provide the level of load balancing that Maglev desires. Second, while minimizing lookup table disruptions is important, a small number of disruptions is tolerable by Maglev. Steady state, changes to the lookup table do not lead to connection resets because connections' affinity to Maglev machines does not change at the same time. When connections' affinity to Maglevs does change, resets are proportional to the number of lookup table disruptions.

With these considerations in mind, we developed a new consistent hashing algorithm, which we call *Maglev hashing*. The basic idea of Maglev hashing is to assign a preference list of all the lookup table positions to each backend. Then all the backends take turns filling their most-preferred table positions that are still empty, until the lookup table is completely filled in. Hence, Maglev hashing gives an almost equal share of the lookup table to each of the backends. Heterogeneous backend weights can be achieved by altering the relative frequency of the backends' turns; the implementation details are not described in this paper.

Let M be the size of the lookup table. The preference list for backend i is stored in $permutation[i]$, which is a random permutation of array $(0..M-1)$. As an efficient way of generating $permutation[i]$, each backend is assigned a unique name. We first hash the backend name using two different hashing functions to generate two numbers *offset* and *skip*. Then we generate $permutation[i]$ using these numbers as follows:

$$offset \leftarrow h_1(name[i]) \mod M$$
$$skip \leftarrow h_2(name[i]) \mod (M-1)+1$$
$$permutation[i][j] \leftarrow (offset + j \times skip) \mod M$$

M must be a prime number so that all values of *skip* are relatively prime to it. Let N be the size of a VIP's backend pool. Its lookup table is populated using Pseudocode 1. We use $next[i]$ to track the next index in the permutation to be considered for backend i; the final lookup table is stored in the array *entry*. In the body of the outer *while* loop, we iterate through all the backends. For each backend i we find a candidate index c from $permutation[i]$ which has not been filled yet, and fill it with the backend. The loop keeps going until all entries in the table have been filled.

The algorithm is guaranteed to finish. Its worst case time complexity is $O(M^2)$ which only happens if there are as many backends as lookup table entries and all the backends hash to the same permutation. To avoid this happening we always choose M such that $M \gg N$. The average time complexity is $O(M \log M)$ because at step n we expect the algorithm to take $\frac{M}{M-n}$ tries to find an empty candidate index, so the total number of steps is $\sum_{n=1}^{M} \frac{M}{n}$. Each backend will take either $\lfloor \frac{M}{N} \rfloor$ or $\lceil \frac{M}{N} \rceil$ entries in the lookup table. Therefore the number of entries occupied by different backends will differ by at most 1. In practice, we choose M to be larger than $100 \times N$ to ensure at most a 1% difference in hash space assigned to backends. Other methods of generating random permutations, such as the Fisher-Yates Shuffle [20], generate better quality permutations using more state, and would work fine here as well.

We use the example in Table 1 to illustrate how Maglev hashing works. Assume there are 3 backends, the

Table 1: A sample consistent hash lookup table.

	$B0$	$B1$	$B2$
0	3	0	3
1	0	2	4
2	4	4	5
3	1	6	6
4	5	1	0
5	2	3	1
6	6	5	2

Permutation tables for the backends.

	Before	After
0	$B1$	$B0$
1	$B0$	$B0$
2	$B1$	$B0$
3	$B0$	$B0$
4	$B2$	$B2$
5	$B2$	$B2$
6	$B0$	$B2$

Lookup table before and after $B1$ is removed.

lookup table size is 7, and the (offset, skip) pairs of the three backends are (3, 4), (0, 2) and (3, 1). The generated permutation tables are shown in the left column, and the lookup tables before and after backend $B1$ is removed are presented in the right column. As the example shows, the lookup tables are evenly balanced among the backends both with and without $B1$. After $B1$ is removed, aside from updating all of the entries that contained $B1$, only one other entry (row 6) needs to be changed. In practice, with larger lookup tables, Maglev hashing is fairly resilient to backend changes, as we show in Section 5.3.

4 Operational Experience

Maglev is a highly complex distributed system that has been serving Google for over six years. We have learned a lot while operating it at a global scale. This section describes how Maglev has evolved over the years to accommodate our changing requirements, and some of the tools we've built to monitor and debug the system.

4.1 Evolution of Maglev

Today's Maglev differs in many details from the original system. Most of the changes, such as the addition of IPv6 support, happened smoothly as a result of the extensible software architecture. This subsection discusses two major changes to the implementation and deployment of Maglev since its birth.

4.1.1 Failover

Maglev machines were originally deployed in active-passive pairs to provide failure resilience, as were the hardware load balancers they replaced. Only active machines served traffic in normal situations. When an active machine became unhealthy, its passive counterpart would take over and start serving. Connections were usually uninterrupted during this process thanks to Maglev hashing, but there were some drawbacks to this setup. It used resources inefficiently, since half of the machines sat idle at all times. It also prevented us from scaling any VIP past the capacity of a single Maglev machine. Finally, coordination between active and passive machines was complex. In this setup, the machines' announcers would monitor each other's health and serving priority, escalating their own BGP priority if they lost sight of each other, with various tie-breaking mechanisms.

We gained a great deal of capacity, efficiency, and operational simplicity by moving to an ECMP model. While Maglev hashing continues to protect us against occasional ECMP flaps, we can multiply the capacity of a VIP by the maximum ECMP set size of the routers, and all machines can be fully utilized.

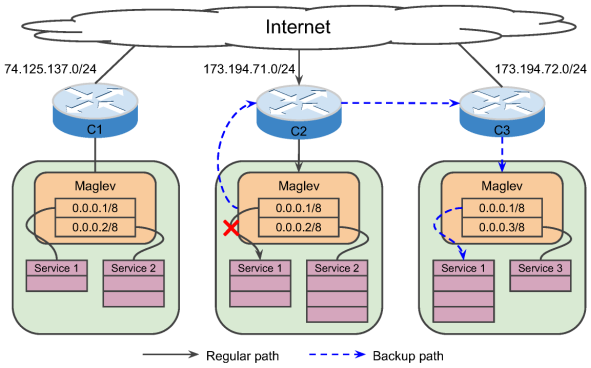

Figure 6: Maglev VIP matching.

4.1.2 Packet Processing

Maglev originally used the Linux kernel network stack for packet processing. It had to interact with the NIC using kernel sockets, which brought significant overhead to packet processing including hardware and software interrupts, context switches and system calls [26]. Each packet also had to be copied from kernel to userspace and back again, which incurred additional overhead. Maglev does not require a TCP/IP stack, but only needs to find a proper backend for each packet and encapsulate it using GRE. Therefore we lost no functionality and greatly improved performance when we introduced the kernel bypass mechanism – the throughput of each Maglev machine is improved by more than a factor of five.

4.2 VIP Matching

In Google's production networks, each cluster is assigned an external IP prefix that is globally routable. For example, cluster C1 in Figure 6 has prefix 74.125.137.0/24. The same service is configured as different VIPs in different clusters, and the user is directed to one of them by DNS. For instance, Service1 is configured as 74.125.137.1 in C1 and 173.194.71.1 in C2.

Google has several different classes of clusters, serving different sets of VIPs. External prefix lengths are the same for clusters of the same class, but may be different for different cluster classes. Sometimes, in emergencies, we need to redirect traffic to a different cluster via Maglev encapsulation. Therefore, we need the target Maglevs to be able to correctly classify traffic for arbitrary other clusters. One possible solution is to define all VIPs in all the clusters that may receive redirected traffic, but that would cause synchronization and scalability issues.

Instead, we implemented a special numbering rule and a novel VIP matching mechanism to cope with the problem. For each cluster class, we assign each VIP the same *suffix* across all clusters of that class. Then we use a pre-

fix/suffix matching mechanism for VIP matching. First, the incoming packet goes through longest prefix matching, to determine which cluster class it was destined for. Then it goes through longest suffix matching specific to that cluster class, to determine which backend pool it should be sent to. In order to reduce the need to keep configs globally in sync on a tight time scale, we preconfigure maglevs with a large prefix group for each cluster class, from which prefixes for new clusters of the same class are allocated. This way a Maglev can correctly serve traffic originally destined for a cluster that it has never heard of.

As a result, each VIP is configured as a <Prefix Group, IP suffix, port, protocol> tuple. Take Figure 6 as an example. Assuming C2 and C3 are of the same class, if a packet towards 173.194.71.1 is received in C2 but Maglev determines none of the endpoints in C2 can serve the packet, it will encapsulate and tunnel the packet towards the VIP address in C3 for the same service (173.194.72.1). Then a Maglev in C3 will decapsulate the packet and match the inner packet to Service1 using prefix/suffix matching, and the packet will be served by an endpoint in C3 instead.

This VIP matching mechanism is specific to Google's production setup, but it provides a good example of the value of rapid prototyping and iteration that a software-based load balancer can offer.

4.3 Fragment Handling

One special case that is not covered by the system described so far is IP fragmentation. Fragments require special treatment because Maglev performs 5-tuple hashing for most VIPs, but fragments do not all contain the full 5-tuple. For example, if a large datagram is split into two fragments, the first fragment will contain both L3 and L4 headers while the second will only contain the L3 header. Thus when Maglev receives a non-first fragment, it cannot make the correct forwarding decision based only on that packet's headers.

Maglev must satisfy two requirements in order to handle fragments correctly. First, all fragments of the same datagram must be received by the same Maglev. Second, the Maglev must make consistent backend selection decisions for unfragmented packets, first fragments, and non-first fragments.

In general, we cannot rely on the hardware in front of Maglev to satisfy the first requirement on its own. For example, some routers use 5-tuple hashing for first fragments and 3-tuple for non-first fragments. We therefore implemented a generic solution in Maglev to cope with any fragment hashing behavior. Each Maglev is configured with a special backend pool consisting of all Maglevs within the cluster. Upon receipt of a fragment, Maglev computes its 3-tuple hash using the L3 header and forwards it to a Maglev from the pool based on the hash value. Since all fragments belonging to the same datagram contain the same 3-tuple, they are guaranteed to be redirected to the same Maglev. We use the GRE recursion control field to ensure that fragments are only redirected once.

To meet the second requirement, Maglev uses the same backend selection algorithm to choose a backend for unfragmented packets and second-hop first fragments (usually on different Maglev instances.) It maintains a fixed-size fragment table which records forwarding decisions for first fragments. When a second-hop non-first fragment is received by the same machine, Maglev looks it up in the fragment table and forwards it immediately if a match is found; otherwise it is cached in the fragment table until the first one is received or the entry expires.

This approach has two limitations: it introduces extra hops to fragmented packets, which can potentially lead to packet reordering. It also requires extra memory to buffer non-first fragments. Since packet reordering may happen anywhere in the network, we rely on the endpoints to handle out-of-order packets. In practice only a few VIPs are allowed to receive fragments, and we are easily able to provide a big enough fragment table to handle them.

4.4 Monitoring and Debugging

We consistently monitor the health and behavior of Maglev as we do any other production system – for example, we use both black box and white box monitoring. Our black box monitoring consists of agents all over the world which periodically check the reachability and latency of the configured VIPs. For our white box monitoring, we export various metrics from each Maglev machine via an HTTP server, and the monitoring system periodically queries each server to learn the latest Maglev serving status details. The system sends alerts when it observes abnormal behavior.

Due to Maglev's distributed nature, multiple paths exist from the router through Maglev to the service endpoints. However, debugging is much easier when we are able to discern the exact path that a specific packet takes through the network. Thus we developed the *packet-tracer* tool, similar to X-trace [21]. Packet-tracer constructs and sends specially marked Maglev-recognizable payloads with specified L3 and L4 headers. The payloads contain receiver IP addresses to which Maglev sends debugging information. The packets usually target a specific VIP and are routed normally to our frontend locations. When a Maglev machine receives a packet-tracer packet, it forwards the packet as usual, while also sending debugging information, including its machine name and the selected backend, to the specified receiver.

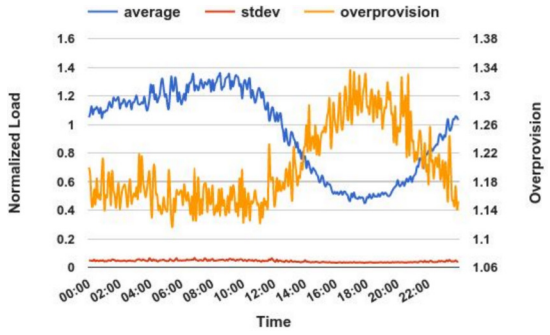

Figure 7: Average, standard deviation and coefficient of variation of normalized load on all service endpoints in one cluster on a typical day.

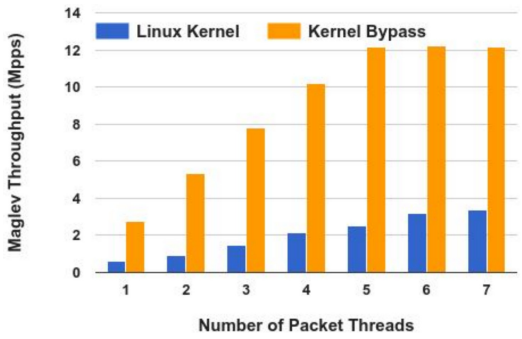

Figure 8: Throughput with and without kernel bypass.

Packet-tracer packets are rate-limited by Maglev, as they are expensive to process. This tool is extremely helpful in debugging production issues, especially when there is more than one Maglev machine on the path, as happens in the case of fragment redirection.

5 Evaluation

In this section we evaluate Maglev's efficiency and performance. We present results from one of Google's production clusters, as well as some microbenchmarks.

5.1 Load Balancing

As a network load balancer, Maglev's major responsibility is to distribute traffic evenly across multiple service endpoints. To illustrate the load balancing performance of Maglev, we collected *connections per second* (*cps*) data from 458 endpoints in a cluster located in Europe. The data is aggregated from multiple HTTP services including Web Search. The granularity of data collection is 5 minutes, and the load is normalized by the average cps throughout the day. Figure 7 shows the average and standard deviation of the load across all endpoints on a typical day. The traffic load exhibits a clear diurnal pattern. The standard deviation is always small compared to the average load; the coefficient of variation is between 6% and 7% most of the time.

Figure 7 also presents the overprovision factor computed as the maximum load over the average load at each time point. It is an important metric because we must ensure even the busiest endpoints will always have enough capacity to serve all the traffic. The overprovision factor is less than 1.2 over 60% of the time. It is notably higher during off-peak hours, which is the expected behavior because it is harder to balance the load when there is less traffic. Besides, a higher overprovision factor during off-peak hours does not require the addition of Maglev machines. This provides a guideline of how much to overprovision at this specific location.

5.2 Single Machine Throughput

Since each Maglev machine receives a roughly equal amount of traffic through ECMP, the overall throughput of Maglev can be estimated as the number of Maglev machines times the throughput of each single machine. The more traffic each machine can handle, the fewer machines will be required to provide the same frontend capacity. Thus single machine throughput is essential to the efficiency of the system.

The throughput of a Maglev machine is affected by many factors, including the number of packet threads, NIC speed, and traffic type. In this subsection we report results from a small testbed to evaluate the packet processing capability of a Maglev machine under various conditions. Unless otherwise specified, all experiments are conducted on servers equipped with two 8-core recent server-class CPUs, one 10Gbps NIC and 128GB of memory. We only use one CPU for Maglev. Everything else, including the operating system, runs on the other CPU. The testbed consists of two senders, two receivers and one Maglev machine located in the same Ethernet domain. The senders slowly increase their sending rates, and the throughput of Maglev is recorded as the maximum number of packets per second (pps)[2] that Maglev can handle before starting to drop packets. We use two senders to ensure Maglev eventually gets overloaded.

5.2.1 Kernel Bypass

In this experiment, we run Maglev in both vanilla Linux network stack mode as well as kernel bypass mode to evaluate the impact of kernel bypass on the throughput of

[2]Note that we report throughput by pps instead of bps because the effect of packet size on the pps throughput is negligible. Hence we measure the pps throughput using minimum-sized packets. The bps throughput is equal to $min(pps \times packet\_size, line\_rate\_bps)$.

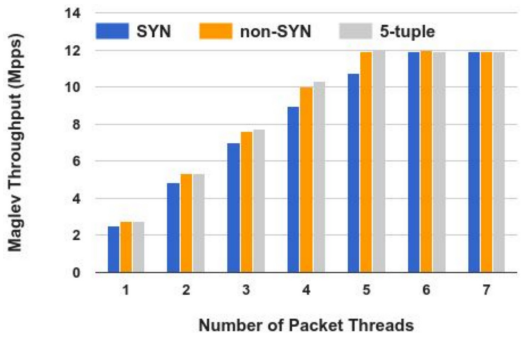

Figure 9: Throughput with different TCP packet types.

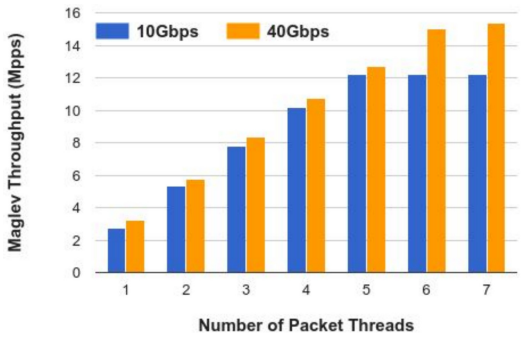

Figure 10: Throughput with different NIC speeds.

Maglev. The senders are configured to send minimum-sized UDP packets from different source ports so that they are not assigned to the same packet thread by the steering module. Due to limitations of the test environment, the minimum size of UDP packets the senders can send is 52 bytes, slightly larger than the theoretical minimum for Ethernet. We vary the number of packet threads in each run of the experiment. Each packet thread is pinned to a dedicated CPU core (as we do in production) to ensure best performance. We use one core for steering and muxing, thus there can be at most 7 packet threads. We measure Maglev's throughput with and without kernel bypass and present the results in Figure 8.

The figure shows the clear advantage of running Maglev in kernel bypass mode. There, Maglev is the bottleneck when there are no more than 4 packet threads; its throughput increases with the number of packet threads. When there are 5 or more packet threads, however, the NIC becomes the bottleneck. On the other hand, Maglev is always the bottleneck when using the vanilla Linux network stack, and the maximum throughput achieved is less than 30% that of kernel bypass.

5.2.2 Traffic Type

Depending on the code execution paths within a packet thread, Maglev handles different types of traffic at different speeds. For example, a packet thread needs to select a backend for a TCP SYN packet and record it in the connection tracking table; it only needs to do a lookup in the connection tracking table for non-SYN packets. In this experiment we measure how fast Maglev handles different types of TCP packets.

Three traffic types are considered: SYN, non-SYN and constant-5-tuple. For SYN and non-SYN experiments, only SYN and non-SYN TCP packets are sent, respectively. The SYN experiment shows how Maglev behaves during SYN flood attacks, while the non-SYN experiment shows how Maglev works with regular TCP traffic, performing backend selection once and using connection tracking afterwards. For the constant-5-tuple experiment, all packets contain the same L3 and L4 headers. This is a special case because the steering module generally tries to send packets with the same 5-tuple to the same packet thread, and only spreads them to other threads when the chosen one is full. The senders vary the source ports for SYN and non-SYN experiments to generate different 5-tuples, but always use the same source port for the constant-5-tuple experiment. They always send minimum-sized TCP packets, which are 64 bytes in our test environment.

As in the previous experiment, Maglev reaches the NIC's capacity with 5 packet threads in the non-SYN and constant-5-tuple experiments. However, for SYN packets, we see that Maglev needs 6 packet threads to saturate the NIC. This is because Maglev needs to perform backend selection for every SYN packet. Maglev performs best under constant-5-tuple traffic, showing that the steering module can effectively steer poorly-distributed packet patterns. Since all packets have the same 5-tuple, their connection tracking information always stays in the CPU cache, ensuring the highest throughput. For non-SYN packets, there are sporadic cache misses for connection tracking lookup, and so the throughput is slightly lower than that for constant-5-tuple traffic when there are fewer than 5 packet threads.

5.2.3 NIC Speed

In the previous experiments, the NIC is the bottleneck as it is saturated by 5 packet threads. To understand Maglev's full capability, this experiment evaluates its throughput using a faster NIC. Instead of the 10Gbps NIC, we install a 40Gbps NIC on the Maglev machine, and use the same setup as in Section 5.2.1. The results are illustrated in Figure 10. When there are no more than 5 packet threads, the 40Gbps NIC provides slightly higher throughput as its chip is faster than the 10Gbps one. However, the throughput growth for the 40Gbps NIC does not slow down until 7 packet threads are used.

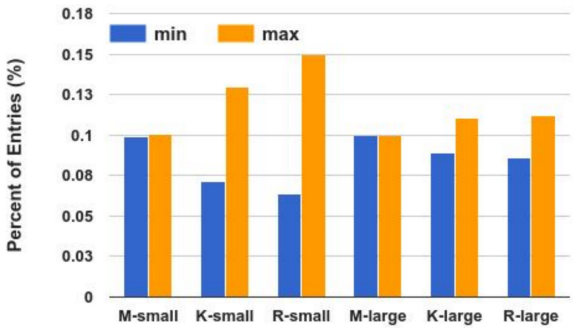

Figure 11: Load balancing efficiency of different hashing methods. M, K and R stand for Maglev, Karger and Rendezvous, respectively. Lookup table size is 65537 for *small* and 655373 for *large*.

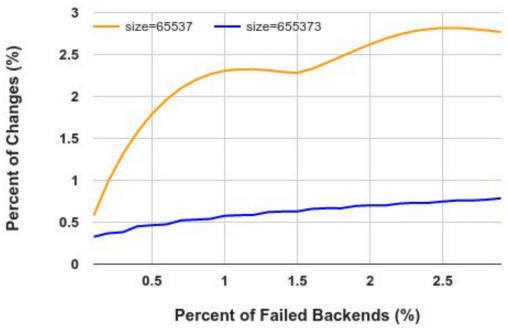

Figure 12: Resilience of Maglev hashing to backend changes.

Because the NIC is no longer the bottleneck, this figure shows the upper bound of Maglev throughput with the current hardware, which is slightly higher than 15Mpps. In fact, the bottleneck here is the Maglev steering module, which will be our focus of optimization when we switch to 40Gbps NICs in the future.

5.3 Consistent Hashing

In this experiment we evaluate Maglev hashing and compare it against *Karger* [28] and *Rendezvous* [38] hashing. We are interested in two metrics: load balancing efficiency and resilience to backend changes.

To evaluate the load balancing efficiency of the methods, we populate one lookup table using each method, and count the number of table entries assigned to each backend. We set the total number of backends to be 1000 and the lookup table size to be 65537 and 655373[3]. For Karger we set the number of views to be 1000. Figure 11 presents the maximum and minimum percent of entries per backend for each method and table size.

As expected, Maglev hashing provides almost perfect load balancing no matter what the table size is. When table size is 65537, Karger and Rendezvous require backends to be overprovisioned by 29.7% and 49.5% respectively to accommodate the imbalanced traffic. The numbers drop to 10.3% and 12.3% as the table size grows to 655373. Since there is one lookup table per VIP, the table size must be limited in order to scale the number of VIPs. Thus Karger and Rendezvous are not suitable for Maglev's load balancing needs.

Another important metric for consistent hashing is resilience to backend changes. Both Karger and Rendezvous guarantee that when some backends fail, the entries for the remaining backends will not be affected. Therefore we only evaluate this metric for Maglev. Figure 12 presents the percent of changed table entries as a function of the percent of concurrent backend failures. We set the number of backends to be 1000. For each failure number k, we randomly remove k backends from the pool, regenerate the lookup table and compute the percent of changed entries. We repeat the experiment 200 times for each k value and report the average results.

Figure 12 shows that the ratio of changed entries increases with the number of concurrent failures. Maglev hashing is more resilient to backend changes when the table size is larger. In practice we use 65537 as the default table size because we expect concurrent backend failures to be rare, and we still have connection tracking as the primary means of protection. In addition, microbenchmarks show that the lookup table generation time increases from 1.8ms to 22.9ms as the table size grows from 65537 to 655373, which prevents us from increasing the table size indefinitely.

[3]There is no special significance to these numbers except that they need to be prime.

6 Related Work

Unlike traditional hardware load balancers [1, 2, 3, 5, 9, 12, 13], Maglev is a distributed software system which runs on commodity servers. Hardware load balancers are usually deployed as active-passive pairs. Maglev provides better efficiency and resiliency by running all servers in active mode. In addition, upgrading hardware load balancer capacity requires purchasing new hardware as well as physically deploying it, making on demand capacity adjustment difficult. On the other hand, Maglev's capacity can easily be adjusted up or down without causing any service disruption. Some hardware vendors also provide load balancing software that runs in virtualized environments. Maglev provides much higher throughput than these virtual load balancers.

Ananta [34] is a distributed software load balancer. Like Maglev, it employs ECMP to scale out the system and uses a flow table to achieve connection affinity.

However, it does not provide a concrete mechanism to handle changes to the load balancer pool gracefully, and it is not specially optimized for single machine performance. Maglev does not have a component similar to Ananta's HostAgent which provides NAT services, but there is an external system (not described here) that offers similar functionality. Ananta allows most internal VIP traffic to bypass the load balancer. Maglev does not provide a similar feature because it has enough capacity for the internal traffic. Embrane [4] is a similar system developed for virtual environments. However, its throughput optimization can be difficult due to the limitations of virtualization. Duet [22] is a hybrid hardware and software load balancer which aims to address the low throughput issue of pure software load balancers. Maglev is able to achieve sufficiently high throughput, thus a hybrid solution becomes unnecessary.

There are also many generic load balancing software packages, the most popular of which are NGINX [14], HAProxy [7], and Linux Virtual Server [11]. They usually run on single servers, but it is also possible to deploy multiple servers in an ECMP group behind a router to achieve the scale-out model. They all provide consistent hashing mechanisms. Compared to Maglev, they mostly prioritize minimum disruption over even load balancing as is done by [28] and [38]. Because they are designed for portability, they are not aggressively optimized for performance.

Consistent hashing [28] and rendezvous hashing [38] were originally introduced for the purpose of distributed cache coordination. Both methods provide guaranteed resilience such that when some backends are removed, only table entries pointing to those backends are updated. However, they don't provide good load balancing across backends, which is an essential requirement for load balancers. On the contrary, Maglev's consistent hashing method achieves perfect balance across the backends at the cost of slightly reduced resilience, which works well in practice when paired with connection tracking. Another option for implementing consistent hashing is distributed hash tables such as Chord [37], but this would add extra latency and complexity to the system.

Some of the performance optimization techniques used in Maglev have been extensively studied since 1990s. Smith et al [36] suggested to improve application throughput by reducing interrupts and memory copying. Mogul et al [33] developed a polling-based mechanism to avoid receive livelock caused by interrupts. Edwards et al [19] explored the idea of userspace networking but did not manage to bypass the kernel completely. Marinos et al [31] showed that specialized userspace networking stacks with kernel bypass can significantly improve application throughput. Hanford et al [25] suggested to distribute packet processing tasks across multiple CPU cores to improve CPU cache hit ratio. CuckooSwitch [41] is a high-performance software L2 switch. One of its key techniques is to mask memory access latency through batching and prefetching. RouteBricks [18] explained how to effectively utilize multi-core CPUs for parallel packet processing.

Several kernel bypass techniques have been developed recently, including DPDK [8], OpenOnload [15], netmap [35], and PF_RING [17], etc. A good summary of popular kernel bypass techniques is presented in [10]. These techniques can be used to effectively accelerate packet processing speed, but they all come with certain limitations. For example, DKPK and OpenOnload are tied to specific NIC vendors while netmap and PF_RING both require a modified Linux kernel. In Maglev we implement a flexible I/O layer which does not require kernel modification and allows us to conveniently switch among different NICs. As with other techniques, Maglev takes over the NIC once started. It uses the TAP interface to inject kernel packets back to the kernel.

GPUs have recently started becoming popular for high-speed packet processing [24, 39]. However, Kalia et al [27] recently showed that CPU-based solutions are able to achieve similar performance with more efficient resource utilization if implemented correctly.

7 Conclusion

This paper presents Maglev, a fast, reliable, scalable and flexible software network load balancer. We built Maglev to scale out via ECMP and to reliably serve at 10Gbps line rate on each machine, for cost-effective performance with rapidly increasing serving demands. We map connections consistently to the same backends with a combination of connection tracking and Maglev hashing. Running this software system at scale has let us operate our websites effectively for many years, reacting quickly to increased demand and new feature needs.

Acknowledgements

We are grateful to Adam Lazur, Alex Tumko, Amin Vahdat, Angus Lees, Aspi Siganporia, Ben Treynor, Bill Coughran, Brad Calder, Craig Bergstrom, Doug Orr, Dzevad Trumic, Elliott Karpilovsky, Jeff Mogul, John T. Reese, Kyle Moffett, Luca Bigliardi, Mahesh Kallahalla, Mario Fanelli, Mike Dalton, Mike Shields, Natalya Etina, Nori Heikkinen, Pierre Imai, Roberto Peon, Simon Newton, Tina Wong, Trisha Weir, Urs Hölzle, and many others for their significant contributions to this paper and the success of Maglev. We would also like to thank our shepherd Nathan Bronson and the anonymous reviewers for their insightful feedback.

References

[1] A10. http://www.a10networks.com.

[2] Array networks. http://www.arraynetworks.com.

[3] Barracuda. http://www.barracuda.com.

[4] Embrane. http://www.embrane.com.

[5] F5. http://www.f5.com.

[6] Google cloud platform. http://cloud.google.com.

[7] Haproxy. http://www.haproxy.org.

[8] Intel dpdk. http://www.dpdk.org.

[9] Kemp. http://www.kemptechnologies.com.

[10] Kernel bypass. http://blog.cloudflare.com/kernel-bypass.

[11] Linux virtual server. http://www.linuxvirtualserver.org.

[12] Load balancer .org. http://www.loadbalancer.org.

[13] Netscaler. http://www.citrix.com.

[14] Nginx. http://www.nginx.org.

[15] Openonload. http://www.openonload.org.

[16] F. Chen, R. K. Sitaraman, and M. Torres. End-user mapping: Next generation request routing for content delivery. In *Proceedings of SIGCOMM*, 2015.

[17] L. Deri. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, 2004.

[18] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of SOSP*, 2009.

[19] A. Edwards and S. Muir. Experiences implementing a high performance tcp in user-space. In *Proceedings of SIGCOMM*, 1995.

[20] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Edinburgh: Oliver and Boyd, 1963.

[21] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of NSDI*, 2007.

[22] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of SIGCOMM*, 2014.

[23] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of SIGCOMM*, 2011.

[24] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of SIGCOMM*, 2010.

[25] N. Hanford, V. Ahuja, M. Balman, M. K. Farrens, D. Ghosal, E. Pouyoul, and B. Tierney. Characterizing the impact of end-system affinities on the end-to-end performance of high-speed flows. In *Proceedings of NDM*, 2013.

[26] V. Jacobson and B. Felderman. Speeding up networking. http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf.

[27] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using gpus in software packet processing. In *Proceedings of NSDI*, 2015.

[28] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of ACM Symposium on Theory of Computing*, 1997.

[29] C. Labovitz. Google sets new internet record. http://www.deepfield.com/2013/07/google-sets-new-internet-record/.

[30] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet inter-domain traffic. In *Proceedings of SIGCOMM*, 2010.

[31] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proceedings of SIGCOMM*, 2014.

[32] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *Proceedings of USENIX ATC*, 2010.

[33] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proceedings of USENIX ATC*, 1996.

[34] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of SIGCOMM*, 2013.

[35] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proceedings of USENIX Security*, 2012.

[36] J. Smith and C. Traw. Giving applications access to gb/s networking. *Network, IEEE*, 7(4):44–52, 1993.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM*, 2001.

[38] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, 1998.

[39] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multi-layer packet classification with graphics processing units. In *Proceedings of CoNEXT*, 2014.

[40] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of SoCC*, 2010.

[41] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of CoNEXT*, 2013.