Flame: Efficient and Robust Hardware Load Balancing for Data Center Routers

Tom Edsall⋆, Andy Fingerhut⋆, Terry Lam†, Rong Pan⋆, George Varghese†

†University of California, San Diego ⋆ Cisco Systems

ABSTRACT

As single TCP flows approach 10 Gbps, *static hash* ECMP load balancing — used by routers today– does a poor job of balancing load in data centers. We describe a new load balancing algorithm, Flame, that is implementable at 480 Gbps with small memory and uses two novel mechanisms. First, Flame uses a Discounting Rate Estimator (DRE); unlike exponential averaging, DRE quickly measures bursts and yet retains memory of recent bursts. Second, Flame binds flows to *hash functions* and not to *paths*. We show Flame is more resilient and efficient than the earlier Flare scheme, and provides better load balancing and is more deployable than Hedera. Flame also allows rebalancing of flows in hardware at rapid rates. This is interesting because we show TCP experiments at 1 and 10 Gbps that demonstrate that recent Linux stacks after 2.6.14 can tolerate rebalancing once every 10 packets with negligible loss of throughput. On the other hand, Windows 2008 stacks have degraded TCP throughput if rebalancing is done more often than 1 in 32,000 packets.

1. INTRODUCTION

To support growth in cloud applications, data centers offer higher aggregate bandwidth by utilizing multiple paths in the network [7, 1, 4]. For example, in the standard fat-tree topology, edge switches load balance across a set of paths to multiple core switches. Effective network load balancing is crucial to allow core network bandwidth beyond that allowable by link technology. For example, today 10 Gbps core links are reasonably priced and 40 Gbps links are expensive. The only way to economically scale large data centers is to load balance traffic across multiple 10 Gbps core links.

While this is a classic trend in networks, the problem is more difficult today because of high-bandwidth edge flows. 10 Gbps has reached the edge; with fast CPUs and adaptors, single TCP flows can approach 10 Gbps. As the number of high-bandwidth edge flows increases, customers are finding that load balancing performance is unsatisfactory for reasons we explain below. Far from being an academic curiosity, router vendors are actively looking to improve the state-of-the-art.

In this paper, we investigate this load balancing problem. In particular, our goal is to spread network load across all available paths [1] in order to realize bandwidth equal to the sum of the paths. However, it is traditionally required that packets within a flow [2] be delivered to the TCP stack in order. If they are not, performance of that connection can suffer, as we quantify in Section 5.4, due to TCP sender congestion window reduction triggered by the reordering.

The standard load balancing algorithm used in routers is *equal-cost multi-path* routing (ECMP) using a *static* hash. ECMP implies that load balancing is done only over equal cost paths, while static hash assigns a flow to a path by hashing the *TCP/IP 5-tuple* with a *single* hash function [3]. Static hashing is computationally fast and requires no state. It also guarantees no reordering of TCP flows as long as paths do not change. While static hash ECMP is universally implemented, it has poor load balancing performance when there are large edge flows, as the following examples demonstrate.

Example 1: Assume we wish to balance 4 flows each with 6 Gbps of bandwidth across 4 equal cost paths of 10 Gbps. The offered load (24 Gbps) is smaller than the network capacity (40 Gbps). Assuming a static hash that distributes uniformly, the probability that all 4 flows will pick distinct paths is only 24/256, less than 10%. Thus with 90% probability, at least two 6 Gbps flows will be assigned to the same 10 Gbps link and thus will be throttled to 5 Gbps each, though one expensive 10 Gbps link is unused! If n flows are uniformly randomly assigned to p paths, each flow is assigned a path with probability $\frac{1}{p}$. The mean number of flows per path is $\frac{n}{p}$ and the standard deviation is $\sqrt{\frac{n}{p}(1-\frac{1}{p})}$.

[1]While port aggregation and multi-pathing are distinct switch features, the forwarding hardware is nearly the same. In this paper, we will refer to them both as *multi-pathing*. A *path* refers to a physical port in port aggregation and a physical path in true multi-pathing.
[2]A *flow* refers to all the packets of a single TCP connection and is identified by a unique 5-tuple.
[3]*5-tuple* is IP source and destination addresses, TCP source and destination port numbers, and the protocol field.

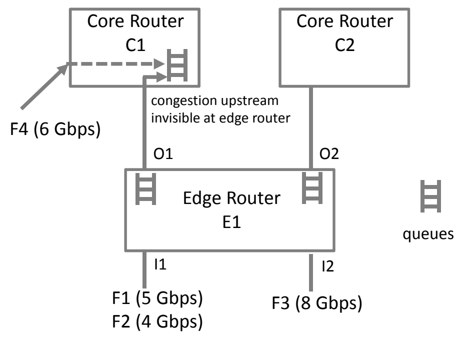


Figure 1: Network topology for Example 2 showing the need to rebalance flows.

As n grows large, the deviation grows as the square root while the mean grows linearly; thus the deviation becomes insignificant as the number of flows increases. But the deviation *is significant* when there are a small number of large flows as in Example 1. Beside random assignment, a second culprit is fixing a flow assignment indefinitely.

Example 2: Assume an edge router (Figure 1) with two input links $I1$ and $I2$ of 10 Gbps, and paths to two different core routers $C1$ and $C2$ via output links $O1$ and $O2$ of 10 Gbps each. Consider three flows, $F1$, $F2$, and $F3$ where $F1$ arrives first on $I1$ and sends at 5 Gbps. A short time later $F2$ arrives on $I1$ and sends at 4 Gbps. Some time later, $F3$ arrives on $I2$ and sends at 8 Gbps. This is a feasible traffic pattern because there is no more than 10 Gbps arriving on any input link. Assume that static hash ECMP gets "lucky" and assigns $F1$ to $O1$ and $F2$ to $O2$. At this point in time, traffic is well balanced. However, when $F3$ arrives, static hash can only assign $F3$ to either $O1$ or $O2$. In either case, we have at most 5 Gbps on one output link and at least 13 Gbps on the other output link. For instance, if $F3$ is assigned to $O1$, then 13 Gbps cannot be sustained on a 10 Gbps output and so queues will build on $O2$ or downstream in core router $C1$. The "right packing" would be to move $F2$ back to $O1$ along with $F2$ and then to assign $F3$ to $O2$.

Related work: Flare [8] goes beyond static hash using two ideas. First, long flows are broken into multiple flowlets based on a packet gap timeout. Second, the first packet of each flowlet is allocated to the least loaded link and the result is stored in a flowlet table and used to route subsequent flowlet packets. A gap timeout larger than the network latency ensures no reordering will occur. In Example 2, if $F1$ and $F2$ are sufficiently spaced apart, $F2$ is *guaranteed* to be assigned to a different link by Flare unlike static hash. Flare does no repacking and will not address the imbalance in Example 2 when $F3$ arrives — unless $F1$ or $F2$ have a sufficiently long gap and are timed out.

Hedera [2] does not attempt to optimally place flows initially. After flows are measured as "heavy-hitters" they are reassigned by a heuristic implemented in *software* on a centralized switch controller. Hedera thus assumes Open Flow [11] or MPLS to control flow routes via software. Hedera allows entire paths to be rebalanced which goes beyond link-by-link balancing as in Flare. However, it not immediately deployable in today's networks by changes to single routers. Further, Hedera rebalances in the order of seconds, implying a few seconds worth of imbalance when a new flow such as $F3$ arrives in Figure 1.

Finally, a reordering-resilient TCP (or an implementation that reorders packets in the destination network adaptor) can allow packet-by-packet load balancing to get near-optimal load balancing. A more approximate alternative is to split large TCP flows into multiple TCP subflows at the source as in Multipath TCP [15]. However, both Hedera and Multipath TCP are clean slate approaches while Flare only requires implementation changes within a single switch. We choose to work in the Flare setting but to go beyond Flare as follows.

Paper contributions: Our contributions include:

1. *New bandwidth estimator:* We propose a new Discounting Rate Estimator (DRE) for link bandwidth that responds faster to new bursts than an exponential weighted moving average (EWMA) while retaining memory of past bursts.

2. *Remembering hash functions not paths:* We use standard power-of-choice hashing to pick the least loaded link which reduces hardware comparison overhead. Unlike Flare, however, we remember the *hash* corresponding to the least loaded link and not the *path*. We show this is more memory-efficient and robust to hash collisions (no worse than ECMP).

3. *Hardware for 48-port 10 Gbps switch:* We use a hash table instead of a per-flow state table to deal with memory overflow; we integrate heavy-hitter detection to maximize efficiency; and (most importantly) how to incorporate periodic load balancing at any parameterized value (even 1 in 10 packets) in *hardware* (Figure 4).

4. *New load balancing metrics:* We introduce new measures for comparing load balancing schemes (Section 4.2 and 5.1).

5. *Updated experiments on the effect of rebalancing on TCP:* In Section 5.4, we describe new experiments with various combinations of Windows and Linux stacks. While Windows stacks degrade considerably in throughput, we show that the latest Linux stacks (after 2.6.14) allow load balancing as often as 1 in 10 packets with at most 10% loss in throughput!

Algorithm 1 Discounting Rate Estimator (DRE)

```
Parameters:
  $T_P$: DRE timer period
  $R_P$: DRE discount ratio
for each path $i$ do
  initialize shallow counter $Q[i] = 0$
end for
loop
  if packet $D$ sent to path $i$ then
    $Q[i] = Q[i] + D.size$
  end if
  if proxy queue timer $T_P$ expires then
    for each path $i$ do
      $Q[i] = Q[i] - Q[i] \cdot R_P$
    end for
  end if
  if $f$ is new flow then
    assign $f$ to path of smallest $Q$
  end if
end loop
```

2. MECHANISMS

In this section, we describe the essential ingredients of our dynamic load balancing scheme. We first describe a *Discounting Rate Estimator* to measure link loads. We then describe our flow table design that enforces packet ordering, show how heavy-hitter detection can improve performance, and finally show how to rebalance flows. These mechanisms are combined in Figure 4.

2.1 Discounting Rate Estimator (DRE)

We start with the Flare approach. Thus, we need a link bandwidth estimator to assign new flows to the least loaded link. Two requirements for a bandwidth estimator are:

Quick reaction to new bursts: In Example 2, if $F2$ arrives a short time after $F1$ and $F1$ has been assigned to output link $O1$, we would like the link estimator for $O1$ to quickly ramp up so that $O1$ looks "more loaded" than $O2$ and $F2$ is (correctly) assigned to $O2$.

Remembering old bursts: In Example 2 again, suppose that $F2$ arrives 100 usec after $F1$ has finished sending at 5 Gbps for a few seconds. At this moment assume that $O1$'s queue is empty and so is that of $O2$. However, the effect of $F1$'s burst may still remain downstream at core router $C1$ in Figure 1. Thus, we would like the path estimator at the edge router to "remember" the fact that $F1$ has sent a burst for a small period equal to the network latency (say 300 usec). Otherwise, $F2$ could wrongly be assigned to $O1$ causing unnecessary congestion at core router $C1$.

Let us see how four standard estimators do with respect to these requirements:

1. Epoch estimator: Bandwidth is traditionally measured by counting how many bytes are sent in a fixed epoch interval (e.g., 1 msec, or 1 second). Then, the counter value is reset and bytes are counted for the next epoch interval. New flows are assigned to the link with the smallest current epoch counter. Unfortunately, the epoch estimator keeps no memory after an epoch. For example, in Example 2, if $F1$ ends just before the end of an epoch, and $F2$ arrives soon after the epoch ends, there will be no memory of $F1$'s burst and $F2$ could be wrongly be assigned to $O2$. One could use the epoch estimator of the last epoch; but such an estimator will not react fast if $F1$ and $F2$ start in the same epoch.

2. Token estimator: The Flare paper [8] uses a token counting approach of subtracting the ideal bytes to be sent on each link from the actual bytes and sending packets to the link with the least tokens. To avoid keeping memory forever, the token counters are periodically reset to zero. This is identical to the epoch estimator in the case of whole flow load balancing and keeps no memory of past bursts across measurement intervals.

3. Exponentially weighted moving average (EWMA): If an EWMA estimator uses a small weight for new information and a large weight for past information, EWMA will keep memory of old bursts but react quite slowly to new arrivals. On the other hand, if we make the weight for recent information to be high and the weight for the past to be low, then the past is forgotten very quickly.

4. Physical queue size: In Figure 1, using the physical queue size of output link $O1$ and output link $O2$ to determine the least loaded link does not work. In Example 2, after flow $F1$ has been assigned to output link $O1$, the physical queue at the edge router is likely to be zero because $F1$ is 5 Gbps and the link is 10 Gbps. But $F1$ can cause congestion at the upper core router $C1$ because of a fourth flow $F4$ that wishes to go on the same path downstream from $C1$. This congestion is invisible at the edge router. Fundamentally, physical queue size does not work because it does not reflect past traffic sent at a rate smaller than the link bandwidth.

To reconcile the simultaneous demands of fast reaction to new bursts while retaining memory of old bursts, we were led to invent a new Discounting rate estimator (DRE) that, to the best of our knowledge, we have not seen before in the literature on estimators. Pseudocode for DRE is shown in Algorithm 1. DRE keeps a counter Q_i for each switch port output link i which is incremented by the packet size when a packet is transmitted on that output link. However, the counter for path i is not periodically reset to zero. Instead, every period say T_p, the counter is decreased by an amount *proportional* to the current counter value. We call the proportionality factor the *discount factor* R_p. If R_p is chosen to be a power of 2, discounting can be implemented in hardware using a shifter and a subtractor. For example, in Section 5 we use very small values of DRE parameters such as $T_P = 100$us and $R_P = 1/512$.

DRE will quickly react to new bursts because it simply

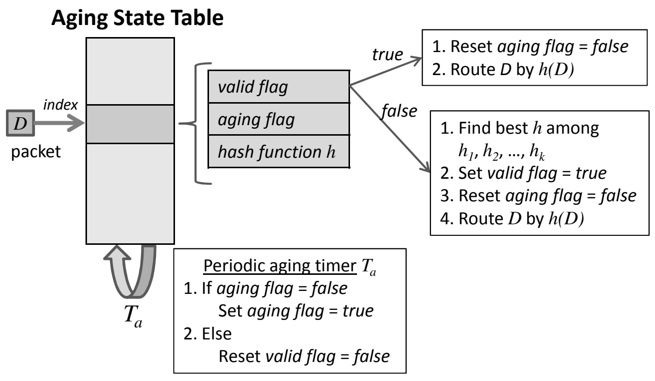


Figure 2: Overview of Flame state table design. A new table entry is set up by comparing k paths as specified by k independent hash functions $h_1, h_2, \ldots, h_k$. The period aging timer is triggered every time interval T_a to age out inactive table entries.

adds the packet bytes. DRE also remember old bursts because every period, the DRE counter is merely discounted by R_p and not reset. The DRE counter will also not diverge to infinity because the higher the counter, the greater the discounting effect. We prove formally in Section 4.1 that the DRE counter stays bounded, is a scaled rate estimator, and balances rise and fall times for new and old bursts. DRE is almost identical to EWMA except that while EWMA weights both old and new information, DRE only weights past information. While this is a simple change, it makes a great difference to rate estimation.

2.2 Choosing the least loaded link

Figure 2 illustrates the key components of the design. In this section, we describe how to choose the least loaded link. This seems trivial. When a new flow F arrives, the forwarding table yields the set of equal cost paths P for F. Next, simply read the DRE counters of all links in P and assign F to the path with the smallest DRE counter. This is unworkable for three reasons:

1. Large number of potential paths: In data centers today, 8 and 16-way multipathing are common but there is growing interest in multi-pathing as high as 32 or even 64. More concretely, consider a fat-tree topology that is maximal in diameter and a top-of-rack switch with 96 ports. With 40 servers in the rack, there will be 40 uplinks which results in 40 ECMP paths

2. Rising traffic rates: 48-ports 10 Gbps Ethernet switches are already in the market. These require a lookup rate of approximately 750 million lookups per second, implying a clock rate of 750MHz. This gives us 1.3nsec per clock cycle which is near the limit of ASIC technology. This allows for only a small number of register reads.

3. Exponential numbers of potential ECMP path sets: If there were 64 ECMP paths used by *all* flows, one could do incremental computation by keeping a pointer to the least loaded link; when a DRE counter is updated, if it is lower than the current lowest the pointer is updated. Unfortunately, each flow can use a different subset of the 40 output links, leading to 2^{40} possible subsets, too many to keep state for, let alone update. Consider the case when an edge router E has 32 outlinks to 32 core routers. One of the core routers, say C, has a failed downlink to an edge router E'. Then, flows from E to E' cannot be routed by the output link to C but the remaining flows can. Similar patterns of failure can result in every possible subset of paths being chosen by some flow.

We cope with the small time budget and the small number of register reads possible using power-of-choice hashing [12]. When a flow first starts, we hash it with k independent hash functions h_1, h_2, $\ldots$, h_k function to get k paths, say P_1, P_2, $\ldots$, P_k. If P_i is the least loaded path, we assign the new flow to P_i. So far this is standard power of choice for load balancing as has been proposed for server load balancing [12].

What is new in our setting is the need to maintain flow order to avoid TCP throughput degradation. Instead of remembering the *path* P_i in a hash table, we remember the hash h_i that generated the least loaded path index. This is a good idea for two reasons. First, we can remember more flows if the state is smaller: the state needed to remember a hash is $\log_2 k$ (2 bits for 4 hash functions!) is much smaller than the 128 bits required to remember a TCP flow. Second, since we do not store the flow ID, then we have to deal with hash collisions. Remembering a hash function is *more robust than remembering a path* because if two flows collide, the second flow will not use the path of the earlier flow but the hash of the earlier flow. Thus the collided flow has a significant chance of being assigned to a different link, no worse than static hash ECMP; on the other hand, we show examples later where remembering paths is much worse than ECMP.

Note that in the special case $k = 2$, we further enhance the computation of two-hash choices so as to guarantee no hash collision as follows. (Otherwise, done independently, there is a $1/4$ probability that the two will pick the same link to sample which is wasteful.) Despite this coupling, the two hash functions are "sufficiently independent" to guarantee good sampling, based on some recent unpublished work by Mitzenmacher.

Let p be the number of equal cost paths. Then given a flow f, we compute its two path choices by the following formulas.

$$P_1 = h_1(f) \mod p \qquad (1)$$
$$P_2 = (h_2(f) \mod (p-1) + 1 + h_1(f)) \mod p$$

2.3 State table design

Figure 2 shows how we keep and update flow state. In particular, when a packet arrives, if the packet's flow is not in the state table, we use the power-of-choice method in Section 2.2 to assign it to the optimal path and insert the chosen hash into the table. The packet is also sent to the path just selected. On the other hand, if the packet's flow is already in the table, then we just send the packet to the path indicated by the stored hash applied to the packet's flow ID.

While the Flare paper [8] indicates that the number of concurrent flowlets is small, their traces were wide-area traces. For large data center traces, we see no reason why the number of concurrent flowlets cannot be much larger. In order to reduce the amount of state to a more affordable level, we use a hash of the flow (instead of exact-matching on the flow ID) to index into the state table. If the flows are hashed into a state table of, say, 1024 buckets, then each bucket can be dynamically assigned a path just as a flow was assigned a path dynamically. To free up state entries as soon as possible, we use a simple and efficient aging mechanism akin to LRU page eviction. We associate an aging field (typically a one-bit flag, but could be larger for finer granularity) with each state table entry and update it when a packet arrives. A complete design of the Flame state table with the aging mechanism is shown in Figure 2. Each table entry is a record with three fields: *1. valid flag*: indicates whether the table entry is valid and contains the state of an active flow; *2. aging flag*: marks inactive or idle flows. *3. hash function*: points to one of the k hash functions $h_1, h_2, \ldots, h_k$.

When a packet arrives, its flowID is hashed to map the packet to a table entry. If the table entry is valid, the packet is dispatched according to the *hash function* stored in the state entry. The aging flag is also cleared to indicate that the flow entry is active. On the other hand, if the state entry is invalid, we make the table entry valid by comparing k paths as specified by k independent hash functions $h_1, h_2, \ldots, h_k$ as in Section 2.2. The hash function resulting in the optimal path is saved in the table entry. Subsequent packets of the same flow are guaranteed to use the same hash function and the same path.

Further, a timer process visits every table entry every *aging* timeout T_a. When it visits a table entry, it either turns on the aging flag or invalidates the entry if the aging flag is already on. In other words, T_a is the timeout threshold to age out inactive flows. Note that with this aging timer process, the aging interval is in the range between T_a and $2T_a$. As suggested in [8], our timing mechanism will split a long flow into several much smaller *flowlets*. While [8] suggests a flowlet timeout of 60 ms based on wide area traces, we suggest that this be a parameter; even numbers like 300 usec may be reasonable in a modern data center.

2.4 Handling heavy-hitters

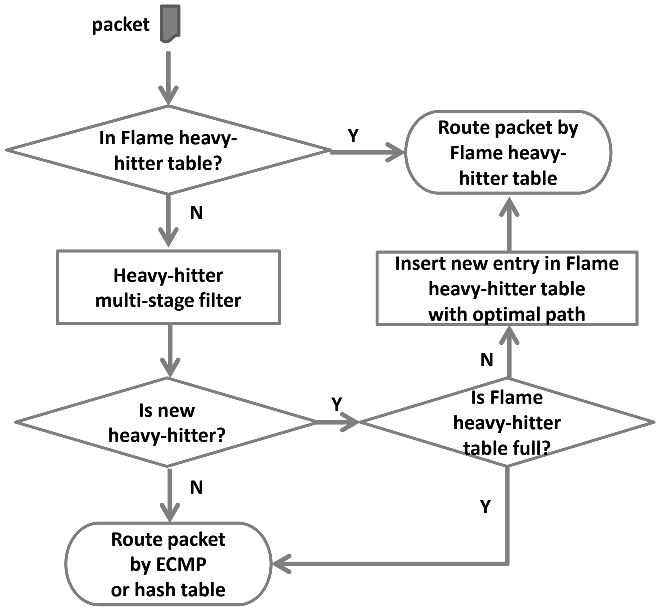

packet
In Flame heavy-hitter table?
Y
Route packet by Flame heavy-hitter table
N
Heavy-hitter multi-stage filter
Insert new entry in Flame heavy-hitter table with optimal path
N
Is new heavy-hitter?
Y
Is Flame heavy-hitter table full?
N
Y
Route packet by ECMP or hash table

Figure 3: Overview of Flame scheme with an exact-matching heavy-hitter table. A flow is routed according to the Flame heavy-hitter table once it is classified as heavy-hitter. Each entry in the Flame heavy-hitter table is also periodically aged out with an aging flag and aging timeout. Non heavy-hitters can use hash state table in Section 2.3.

The hash scheme allows collisions and a flow $F2$ can reuse the state set up by an earlier flow $F1$. If $F1$ is a low rate flow, while $F2$ sends at 5 Gbps, $F2$ will be routed by a static hash but this is not optimal. If $F1$ or $F2$ keep sending, $F1$'s entry will never be timed out and $F2$ will keep using essentially ECMP instead of a more optimal assignment. In this section, we seek to combat this problem by applying the load-balancing algorithm only to *heavy-hitters* since only a small number of heavy-hitters are responsible for a large fraction of traffic in the network [7, 3]. Non-heavy-hitters (i.e. mice flows) can be forwarded using hash state table in Section 2.3 or ECMP.

We also want to allow hardware rebalancing unlike the Hedera paper [2] which only allows rebalancing in software. Since heavy-hitter admission and eviction can introduce another form of TCP packet reordering, we need a parameter F (reordering parameter) to decide how often a flow can be reordered without causing undue harm to TCP. For example, a prior experimental study [9] suggests that TCP performance will not be adversely effected if the number of reorderings is less than 0.1%, i.e. $F = 1000$. Our updated study reported in Section 5.4 suggests that F can be much worse $(32,000)$ for Windows Server destinations and much better (even $F = 10$) for recent Linux versions. Thus we leave F as a parameter. We keep a packet counter for each heavy-hitter in the heavy-hitter table. Every time the packet counter reaches multiple of F, we set up a rebalancing flag which allows the heavy-hitter to be reassigned.

We employ a heavy-hitter multistage filter algorithm with conservative update of counters as in [6]. There are four heavy-hitter detection tables. Each table consists of 1024 counters. A packet is indexed into these tables by four different hash functions. The counters are updated with the packet size according to the conservative update rule. If the counters at *all* four tables exceed a threshold B_H, the flow is classified as a heavy-hitter. Typically $B_H = 3$ KBytes in our experiments. The counters are reset to zero every T_H interval. Typically $T_H = 30$ msec in our experiments.

If a flow passes the heavy-hitter filter, it will be admitted into the heavy-hitter table and switch the load balancing policy from ECMP/hash-table to heavy-hitter based Flame. We can choose *graceful insertion* by initially setting the path to be the same as ECMP and not that of the least loaded path. Alternately, as in our experiments, we can choose to use the least loaded path when the heavy-hitter is first inserted. Such *abrupt insertion* can cause a reordering when the the heavy-hitter is first detected while graceful insertion will not.

We propose the following eviction policy: evict a flow if it sends less than B_H bytes in k_e consecutive periods of T_H. In our experiments, typically $k_e = 3$. When a heavy-hitter is evicted, its state can be immediately deleted from the heavy-hitter table (*abrupt eviction*), and subsequent packets of that flow routed under ECMP. Clearly abrupt eviction can result in packet reordering. Instead, in *graceful eviction*, when the heavy-hitter traffic is below a threshold, we turn on an *eviction-ready* flag and start counting the remaining packets. Then we only physically delete it from the heavy-hitter table upon either *i)* next flowlet aging expiry or *ii)* packet count $> F$. Graceful eviction avoids reordering and is effective against both inactive and slow heavy-hitters. We can tune to accept fewer heavy-hitters by increasing B_H and/or decreasing T_H.

2.5 Profile-based rebalancing

Rebalancing a heavy-hitter by greedily selecting the least utilized path is vulnerable to the following *greedy flash crowd* effect. Consider ten heavy-hitter flows and three paths P_1, P_2, and P_3. The best path assignment is by having three flows to path P_1 and P_2 each and four flows to path P_3. Let's denote this path assignment by a triple $(3, 3, 4)$, i.e. each number in the triple denotes the number of heavy-hitters being assigned to the respective path. Since our greedy method in Section 2.2 is not perfect, typically we only get a near-optimal assignment such as $(2, 4, 4)$. Now if the heavy-hitters are rebalanced frequently, with the initial path assignment $(2, 4, 4)$, they all will be reassigned to path P_1, leading to the subsequent path assignment $(10, 0, 0)$. Next,

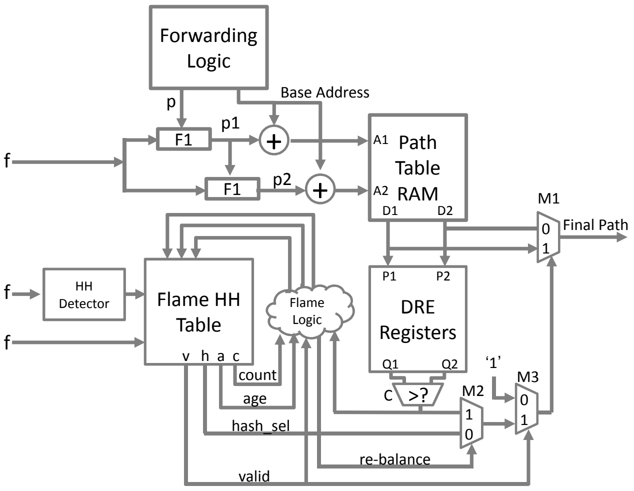


Figure 4: Flame hardware schematic

they all will move away from path P_1, leading to the path assignment $(0, 10, 0)$, and so on. Clearly, such oscillation is undesirable.

We propose the following traffic profiling approach to mitigate this problem. First, we profile heavy-hitter traffic by having a counter per heavy-hitter that counts heavy-hitter bytes in the *previous* epoch. Second, we also maintain a path profile, which is initialized to the path traffic in the *previous* epoch. Third, if we need to rebalance a heavy-hitter, we will rely on both the heavy-hitter profile and the path profile. We ensure that the heavy-hitter reassignment only *improves* the path profile. Finally, we also adjust the path profile by the amount of the flow profile at the rebalancing moment.

As an example, suppose we have three paths with path profile (P_1, P_2, P_3) = (8 KBytes, 9 KBytes, 11 KBytes). If the heavy-hitter were from path P_3 with heavy-hitter profile 2 KBytes, we would rebalance it to path P_1 and update the path profile to (10 KBytes, 9 KBytes, 9 KBytes). However, if the heavy-hitter were from path P_2 with heavy-hitter profile 2 KBytes, we would still keep it at path P_2 since moving to path P_1 would lead to an even worse state.

3. HARDWARE IMPLEMENTATION

Figure 4 describes a hardware block diagram for a chip we are building that puts together all the mechanisms we described earlier including hardware rebalancing every F packets.

Start at the top of Figure 4. The forwarding logic provides a base address into the path table and the number of paths p. The flow ID f is hashed using a hash function $F1$ but that is modified (see Equation 1) in the lower path to essentially compute a second hash function. This produces two offsets $p1$ and $p2$ that are added to the base address and used to index in parallel into a dual-ported memory using addresses

$A1$ and $A2$. The two addresses $A1$ and $A2$ yield two link IDs $D1$ and $D2$ from the path table. (This level of indirection allows graceful handling of path failures). The two link IDs are used to index into the DRE registers to produce two DRE values $Q1$ and $Q2$. Comparator C picks the least loaded link of the two and outputs the result to multiplexor (mux) $M2$.

Now move to the bottom of the figure. Concurrently, the flow ID f is also fed to the Flame table whose output is four values: a valid flag v, a hash select flag h (since we use only two hashes for power of choice, 1-bit suffices), and age bit a, and a count c (an integer of at least 17 bits capable of counting to $32,000$). If the valid flag is "false" (the flow has no valid entry), the mux $M3$ will select the input 1 and pass it as the selection value for mux $M1$, (note that when the selection bit shown at the bottom of a mux is 1, the output corresponds to the input labeled 1, and vice versa). In this case, mux $M1$ picks the output link $D1$ and the forwarding is exactly as in static hash ECMP. This is correct because when there is no state for f we should use ECMP.

If the valid flag is "true" (the flow entry is valid), the mux $M3$ will select the input 0 which is the output of mux $M2$. This value from $M2$ is then fed to $M1$ to select the proper output link, either $D1$ or $D2$.

When the "re-balance" signal is 0, the $M2$ mux will select as its output the hash select signal coming from the Flame table. On the other hand, if the re-balance signal is a 1, the mux $M2$ selects as output the least loaded link from the output of comparator C as we described above. This is fed via mux $M3$ to set the select signal for $M1$ which now actually selects the least loaded path as the output of mux $M1$, The rebalance signal is computed by the (simple) Flame logic. If either the age is 0 or the $count > F$, the rebalance signal is asserted. At the same time, the least loaded link output of comparator $C1$ is fed back via the Flame logic to be stored in the Flame Table.

Note that unlike the Hedera paper in which software periodically identifies heavy-hitters and moves flows to paths, the entire rebalancing process is done in hardware. This is essential if one wishes to do fine-grain rebalancing say once every 10 or 100 packets which is possible without significant TCP degradation for Linux servers as we show in Section 5.4.

4. ANALYSIS

We analyze the *Discounting Rate Estimator (DRE)* in Section 2.1. and then present simple theorems about the robustness of Flame in Section 2.3.

4.1 DRE analysis

Let T_P and R_P be the timer period and discount ratio parameters for DRE. Let $q(t)$ denote the value of the DRE counter at time t. Our DRE model is described by the following differential equations:

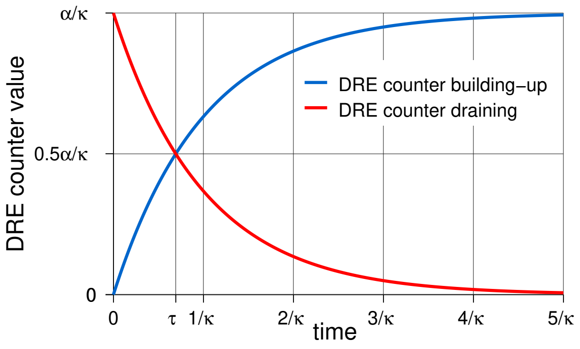


Figure 5: Convergence of DRE counter under constant traffic arrival rate α followed by an abrupt stop to traffic. κ denotes the instantaneous DRE decay rate. τ is the intersection point of the two scenarios.

$$\frac{dq(t)}{dt} = \alpha - \kappa \cdot q(t) \tag{2}$$

where α is the instantaneous traffic arrival rate and κ is the instantaneous DRE decay rate.

Note that with very small value of T_P, we compute the instantaneous decay rate as $\kappa \approx R_P/T_P$. Suppose α is a constant for all time t, we can solve equation (2) as follows.

$$\frac{d(\alpha - \kappa q(t))}{\alpha - \kappa q(t)} = -\kappa dt$$
$$\ln(\alpha - \kappa q(t)) = -\kappa t + \gamma'$$
$$q(t) = \frac{\alpha - \gamma e^{-\kappa t}}{\kappa} \tag{3}$$

where γ and γ' are constants determined by initial conditions

Equation (3) indicates that when a flow has been sending at rate α for a while and stops sending, its DRE counter starts from the stabilized value α/κ and then decays exponentially to 0. On the other hand, when a fresh flow starts sending at rate α, its DRE counter starts from 0 and increases exponentially to the stabilized value α/κ. The exact equations for these two scenarios are as follows.

DRE counter building-up: Suppose a flow has not been sending before time 0^- and then starts sending at time 0^+ with a rate α. Then the boundary conditions are $q(0) = 0$ and $\gamma = \alpha$. Hence, its DRE counter value can be described by:

$$q_1(t) = \frac{\alpha - \alpha e^{-\kappa t}}{\kappa} \tag{4}$$

DRE counter draining: Suppose a flow has been sending at rate α up to time 0^- and then it stops at time 0^+. The boundary conditions are $q(0) = \alpha/\kappa$ and $\gamma = -\alpha$. Hence, its DRE counter can be described by:

$$q_2(t) = \frac{\alpha e^{-\kappa t}}{\kappa} \quad (5)$$

Stabilizing point of DRE counter: Figure 5 illustrates the DRE counter building-up and draining scenarios. Let τ denote the intersection time. By solving $q_1(\tau) = q_2(\tau)$, we get

$$\tau = \ln(2)/\kappa \quad (6)$$

Since network traffic is not continuous but consists of discrete datagram packets, we also verified our model using Matlab simulations with practical data center settings (e.g. bandwidth 10 Gbps and 20 Gbps). In particular, we validated two important properties of the DRE design: the cross point τ is independent of the arrival rate α and $\tau = \ln(2)/\kappa$. From Figure 5, we observe that DRE counters are bounded and eventually converge as long as the arrival rate is bounded. Further, the DRE counters converge quickly as measured by the metric τ and so the DRE timer period should be larger than τ.

Note the important property that the parameter κ itself defines where the cross point is, and is independent of the arrival rate α. In the context of load balancing flows, we believe that τ should be in the order of the queuing delay in the network. In particular, we can set the DRE parameters according to the formula $T_P \times R_P = d$ where d is the network delay.

4.2 Analysis of Flame state table design

In this section, we analyze Flame, particularly the hash-table based approach of Section 2.3. We argue that Flame is robust and outperforms Flare in general and yet there are certain circumstances in which Flame degenerates to ECMP but in which Flare does poorly.

Our notation is as follows. Let k be the number of hash functions in Flame (Section 2.2). Let p be an upper bound on the number of equal paths individually denoted as P_1, $P_2, \ldots, P_p$. Let n be the number of heavy-hitters, individually denoted as $H_1, H_2, \ldots, H_n$. Let m be the number of entries in the state table. Let T_a be the aging timeout. We assume the finite memory versions of Flame and Flare. Recall that Flame remembers one of k hash functions, while Flare one of p paths. Any state table entry is timed out after at most $2T_a$ because of the LRU approximation. When Flare or Flame insert a flow into a hitherto invalid entry, the algorithm measures the current state of all paths and places the flow in the least loaded path: we sometimes refer to this as "sensing" in what follows.

First, observe trivially that if $k = 1$, Flame behaves exactly like ECMP. Next, our first theorem shows that while sensing and assigning to the least loaded path appears to be a good idea it can sometimes backfire when there are bursts. That means least loaded path schemes do not always outperform ECMP.

THEOREM 1. **(Burst vulnerability)** *Under bursty arrivals of heavy-hitters, any scheme that allocates a new flow to the least loaded path and preserves flow packet order can perform much worse than ECMP for arbitrary time periods.*

Proof: Consider the following traffic scenario for Flare. Suppose the amount of memory m is much larger than the number of heavy-hitters n so that all heavy-hitters hash to distinct entries in the hash table, with no hash collisions. Without loss of generality, suppose the first heavy-hitter H_1 is assigned to path P_1. Then bring on the second heavy-hitter H_2 after enough time for our load measuring algorithm to "sense" H_1. Without loss of generality, suppose H_2 is assigned to path P_2. Repeat until H_1 through H_{p-1} have been assigned to paths P_1 to P_{p-1} respectively. Now bring on simultaneously H_p , $H_{p+1}, \ldots, H_n$. Since at the start the new heavy-hitters have not sent any traffic, this last burst of heavy-hitters will be assigned to path P_p. Now suppose each of the heavy-hitters continue to send traffic for arbitrary time. Then none of the entries assigned to the heavy-hitters will time out. Thus all will have valid entries, and the system will never sense the links for the least loaded link and reassign because no new flows arrive. However, that means for an unbounded period of time, $(n - p + 1)$ heavy-hitters are assigned to path P_p and one heavy-hitter apiece of P_1 through P_{p-1}, leading to an unbounded load discrepancy over any time scale. $|n - p|$ can be made arbitrarily large by increasing n).

Flame is susceptible to the same "flash crowd" scenario but its imperfect sensing actually makes it somewhat more likely to spread flows out better. For example, if $k = 2$ and p is large, when a later heavy-hitter comes, the probability that neither of the two hash functions picks path P_p is $(1 - \frac{1}{p})(1 - \frac{1}{p-1}) = 1 - \frac{2}{p}$, i.e. $= 75\%$ with $p = 8$. So Flame places 25% of the heavy-hitters on P_p. On the other hand, ECMP would put roughly $1/8 = 12.5\%$. □

The flash crowd scenario of this theorem has two implications. First, it shows all sensing schemes are vulnerable to flash crowds where flows arrive simultaneously and can do worse than ECMP though Flame is better than Flare. This and Example 2 in the introduction suggests that periodic rebalancing is not merely a desirable but a requirement. Second, it shows why heavy-hitter detection can help even in the case of flash crowds if there is sufficient memory to keep state for each heavy-hitter. Note that if there are more than 1000 flows, static hash ECMP should work well because the standard deviation falls with the square root of the number of flows. Thus keeping state for around 1000 heavy-hitters should alleviate this scenario for either Flare or Flame.

Next, we show that remembering paths in Flare can cause robustness problems for Flare but not for Flame. The problem can arise due to *spoilers*, small flows that capture hash table entries early.

THEOREM 2. **(Spoiler resilience)** *Flame is resilient to the presence of spoilers and is no worse than ECMP. However, Flare can be arbitrarily worse than ECMP.*

Proof: Flame resorts to one more level of hashing within a table entry, so degrades gracefully to ECMP in the presence of spoilers. Now consider Flare in the following scenario. We first bring on the first $p-1$ heavy-hitters, well-spaced out in time so as to get assignment in paths P_1 through P_{p-1}. Then, we bring on $O(m)$ spoilers that capture all remaining entries that are left invalid. Since the spoilers are small, they are all assigned to path P_p. Thus, all state table entries are marked as valid, in which $p-1$ entries are assigned to paths P_1 through P_{p-1} and the remaining $m-p+1$ cells are assigned to path P_p. In fact, the system is reasonably load balanced at this time.

Next, we bring on the remaining heavy-hitters H_p, H_{p+1}, $\ldots$, H_n nicely spaced in time so that none is bursty. If m is large, the majority of the state table is filled with path P_p and valid bit set. Thus, the later heavy-hitter will likely pick such a "spoiler entry" and be then assigned to path P_p. Thus with high probability, all later heavy-hitters will be assigned to path P_p. If all heavy-hitters continue sending for an arbitrary period of time, the situation will persist and no further sensing will take place because no entry times out. In other words, we now have $(n-p+1)$ heavy-hitters to path P_p and only one each assigned to paths P_1 through P_{p-1}. So by increasing n without bound, we have arbitrarily bad average and worst case load discrepancy. Even if the spoilers stop sending traffic completely after a heavy-hitter arrives in their cell, the heavy-hitter will keep the cell from timing out even though the recorded path information is prehistoric. $\square$

Doing no worse than ECMP is a good robustness guarantee for Flame. However, since we aspire to *do better* than ECMP, it is better to avoid collisions for large flows (as far as possible) using heavy-hitter filters.

5. EVALUATION

In this section, we experimentally study how Flame does with respect to ECMP and Flare. Unfortunately, we have no access to data center traces; hence, we enhance realistic Internet traces with artificial heavy-hitters, the number, duration, and intensity of which we can control for. This is because realistic and available Internet traces do not have the heavy-hitters required for load balancing to be an interesting; at the same time, purely synthetic traffic appears to be too contrived. We also show results of benchmarking TCP performance under packet reordering at 1 and 10 Gbps which inform the choice of the hardware rebalancing parameter F in Figure 4.

5.1 Load balancing goodness metrics

We start with metrics for load balancing effectiveness. Denote by T_s the measurement time scale parameter. First, fix a value of T_s and divide the traffic trace into disjoint and contiguous time intervals of length T_s. Then for each interval, measure the *path traffic vector* accumulated during the interval and compute the *balancing quality* within the interval using a load balancing *goodness metric* as shown below. Let p be the number of paths and $P_1, \ldots, P_p$ be paths. For path P_i $(1 \le i \le p)$, denote $P_i.load$ as the network traffic on path P_i during the current time interval. We denote average load on all paths as $\overline{P}.load = \frac{1}{p}\sum_{i=1}^{p} P_i.load$. We consider $\overline{P}.load$ to be the ideal balance in the current time interval. Then we propose three goodness metrics:

1. Absolute deviation: worst case bandwidth difference of one path from the ideal balance

$$G_d \triangleq \max_{P \in \{P_1,\ldots,P_p\}} \frac{|P.load - \overline{P}.load|}{T_s}$$

2. Normalized deviation: percent of bandwidth difference from the ideal

$$G_n \triangleq \max_{P \in \{P_1,\ldots,P_p\}} \frac{|P.load - \overline{P}.load|}{\overline{P}.load}$$

3. Jain's fairness index: standard fairness metric for a set of p load values

$$G_{\mathcal{J}} \triangleq \frac{(\sum_{i=1}^{p} P_i.load)^2}{p \cdot \sum_{i=1}^{p} (P_i.load)^2}$$

Note that load balancing quality is better with smaller deviation value and higher Jain's fairness index. Next, the goodness values in all intervals of size T_s form a time series for which we can calculate statistics such as max, average, and $99-th$ percentile. One final complication remains: what is a good choice of T_s?

Flare [8] picks $T_s = 300$ms since that is about the amount of data that a router can buffer. However, in recent data center routers at 10G, even 10 ms worth of buffering is large and the amount of buffering per link may actually decrease further at 40 Gbps. Further, load balancing goodness metrics appear better with larger T_s. As an example, suppose with $T_s = 1$ ms and $p = 3$ paths, we have the following path traffic vectors for (P_1, P_2, P_3) in three successive measurement intervals (100, 0, 0) KBytes, (0, 100, 0) KBytes, and (0, 0, 100) KBytes. Clearly, load balancing performance is poor with absolute deviation $= 67$ MBytes/s. However, by enlarging T_s to 3ms, we have a single path traffic vector (100, 100, 100) KBytes, which apparently has perfect performance and an absolute deviation $= 0$.

Therefore, we visualize load balancing quality across *all* measurement time scales by plotting a graph with the goodness statistic on the y-axis versus time scale choice on the x-axis. Since computing goodness for all choices of T_s is infeasible computationally, we limit T_s to powers of 2 beyond one packet transmission. In particular, in our experiments we only use $T_s = 1, 2, 4, 8, 10, 20, 40, 80$ msec.

5.2 Simulation setup

We use an Internet backbone trace provided by CAIDA [16]. The trace is collected at a San Jose monitor point in 2008 with bandwidth about 1.8 Gbps and length about one minute. We simulate load balancing schemes (ECMP, Flare, and Flame). in Perl using the CoralReef software suite [5]. To impose synthetic traffic on top of any real *pcap* trace, we augment the software to support synthetic events (such as enqueuing and dequeuing synthetic packets at synthetic timestamps) The synthetic events are managed by an efficient implementation of a heap-based discrete-event simulation engine.

The number of heavy-hitters is an input parameter. Each heavy hitter comprises of start time, end time, and traffic pattern. We simulate a heavy-hitter as a large constant-bit-rate FTP file transfer of 50-128 MB. based on the default Hadoop block size [3, 7]. The start time of each parameter is a parameter that can be controlled. For example, we can simulate simultaneous arrival of flows to or have the start time be sampled from a specified distribution.. The end time is either determined by fixing the duration of heavy-hitter (say, 10 seconds) or by randomizing either the duration (e.g. as a Gaussian distribution with mean 10 seconds) or the rate of a heavy-hitter.

Each heavy-hitter is represented by a random TCP 5-tuple. Static hash ECMP does badly when the heavy-hitters have the same source and destination IP address. Finally, our framework allows the simulation of sophisticated heavy-hitter traffic patterns to exhibit several burstiness and flowlet behaviors including ON-OFF heavy-hitters with a Pareto distribution for the OFF period. Such patterns exercise the load balancing algorithms ability to continually admit and evict flows. The flowlet behavior can be controlled; even when a heavy-hitter is OFF, it can send at least one packet every flowlet timeout so that eviction only occurs under the "once every F packets rule". We also allow the introduction of spoilers that capture a hash table bucket and send at a slow rate.

5.3 Simulation results

In the following set of experiments, we impose 8 synthetic flows on the CAIDA Internet trace. The full CAIDA trace lasts for 52 seconds. Each synthetic flow sends at 100 Mbps by dispatching packets of size 1250 bytes at 100 us intervals. The starting times are staggered at 1 second apart, but added random noise up to ± 100 ms. We let the synthetic flows run until experiment completion so that we can observe their full impact. We limit the Flame table to 2048 and overflow to ECMP if the table is full. Our Flame scheme use the heavy-hitter *abrupt admission* and *graceful eviction* policies as discussed in Section 2.4. Unless otherwise stated, the default parameters are as follows: number of paths $p = 3$, heavy-hitter filter threshold $B_H = 3$ KBytes, heavy-hitter filter timeout $T_H = 30$ msec, flowlet aging timeout $T_a = 30$ msec.

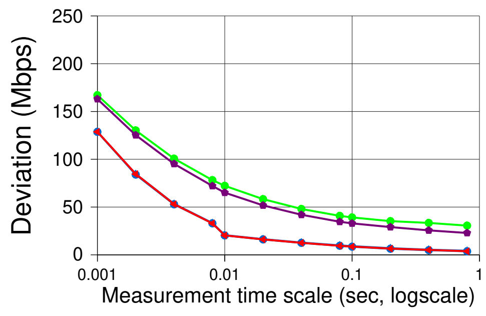
Figure 6 and 7 compare the load balancing performance of ECMP, Flare, and Flame load balancing schemes. Note that higher Jain's fairness index means better fairness quality, which is opposite to other deviation metrics. We also tease apart the effect of each individual Flame mechanisms. For example, we illustrate how the performance changes with the addition of each Flame mechanisms, i.e. a heavy-hitter filter to track heavy-hitters, periodic rebalancing with "1 every 1000 packets rule" and profiling to prevent greedily rebalancing.

We observe that the synthetic heavy-hitters have a significant effect on ECMP and Flare but much less on Flame with a heavy-hitter filter. To be sure, Flare would also improve with a heavy-hitter filter: the real reason for Flame over Flare is the robustness and memory efficiency caused by remembering the hash and not the path. In the figures, we call the "1 in 1000" packets rule the TCP 0.1% rule. Note that Flame with the "HH + TCP 0.1%" curves can be worse than the"HH" only curves because of rebalancing oscillation which is removed by the "HH + TCP 0.1% + profiling" rule.

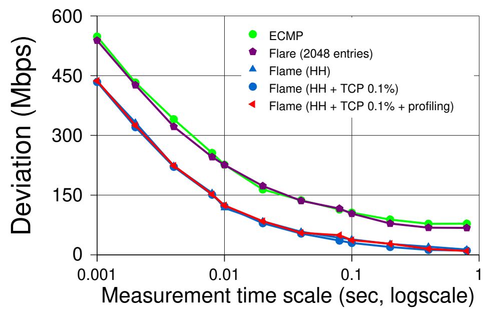
In Figure 6, there is little difference between static hash and Flare while the results of Flame algorithm are better at all time scales. The relatively poor performance of the Flare algorithm is likely due to its state table becoming saturated. Note that while our analytical results went further and suggested that Flare can do *worse* than ECMP, the experimental scenario seems more plausible. Again, the difference is the heavy-hitter filter which, to be fair, would improve Flare as well. However, the experiments do point to the crucial need for robustness and graceful degradation when the memory does not suffice.

Figure 7 shows performance when synthetic traffic is added to the real trace. Here we see a large difference at all time scales between static hash, Flare and Flame with static hash much worse than shown in Figure 6. The Flare results are somewhat worse than without the synthetic traffic although noticeably better than static hash.

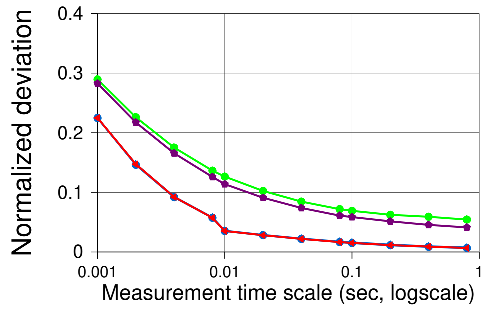
While Flare (without heavy-hitters) performs better than static hash because it is breaking the synthetic flows into flowlets and balancing each one, by separating heavy-hitters Flame does not suffer from table saturation the way Flare does. This allows Flame to intelligently balance a much larger fraction of traffic. These results clearly show that



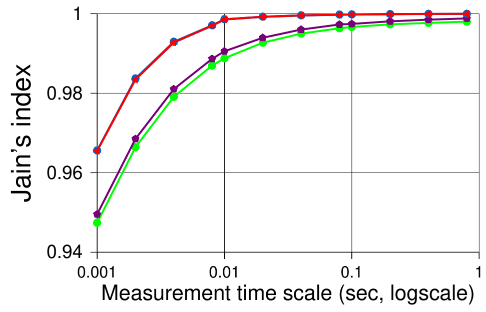
(a) Mean absolute deviation



(b) 99th absolute deviation



(c) Mean normalized deviation



(d) Mean Jain's fairness index

Figure 6: Comparison of load balancing schemes with goodness metrics across all measurement time scales. *HH* denotes inclusion of a heavy-hitter table. Simulation on the CAIDA trace

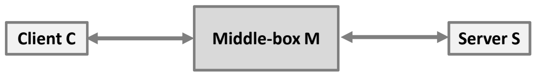


Figure 8: Test-bed for TCP packet reordering

Flame outperforms both static hash and Flare with the amount of state being about the same as Flare. We also evaluated other synthetic heavy-hitter pattern (larger number of heavy-hitters, simultaneous heavy-hitter launch, and rate-varying heavy-hitters) which showed that Flame is resilient to all such combinations. We do not include these graphs due to space constraints.

5.4 Impact of packet reordering on TCP

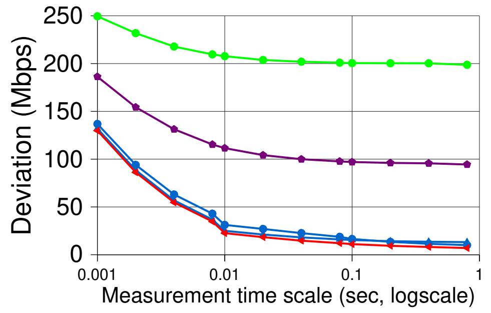
Recall that Figure 4 has a parameter F that controls the frequency of reordering. While earlier studies have suggested $F = 1000$ we felt it was essential to update these studies to see the effects of operating system changes, higher link speeds, and the subtle difference in reordering patterns caused by load balancing compared to arbitrary reordering. To evaluate the impact of packet reordering on TCP performance at 1 Gbps and 10 Gbps, we set up two hardware test-beds, each consisting of three nodes connected serially as shown in Figure 8. The middle-box M controls the forwarding of all traffic between the client C and the server S.

Results from the 1 Gbps test-bed:

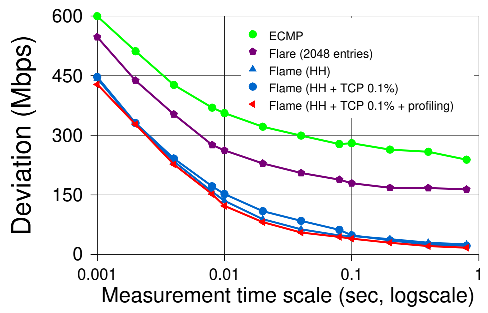
In the 1 Gbps experiments, the middle-box M was a 2-processor Intel Xeon 2.4 GHz machine running Ubuntu 10.10 32-bit server with Linux kernel 2.6.35. The kernel was recompiled after applying the Trace Control for Netem patch [13, 17] which enables the flexible addition of latency to packets needed for our experiments. The client and server used the same model of machines as the middle-box. For Linux client and server experiments, they were running Ubuntu 8.10 32-bit server with Linux kernel 2.6.27. For Windows experiments, they were running Windows Server Standard 2008, 32-bit, SP2. All machines had two Intel PRO/1000 MT Desktop Ethernet interfaces, but only one was enabled on the client and server machines.

In the 1 Gbps experiments, the middle-box was configured to reorder packets by selecting a number of packets F. The first $F/2$ packets sent out of a network interface had 0.9 msec of extra latency added to the normal time required to forward the packet. The next $F/2$ packets sent had 1.1 msec of extra latency added. This pattern repeats every F packets, so when the latency changes from 1.1 msec to 0.9 msec, one or more packets can be transmitted out of order. This emulates the situation where a flow's packets are switched from a low to high latency path, then switched from a high to low latency path, every F packets.

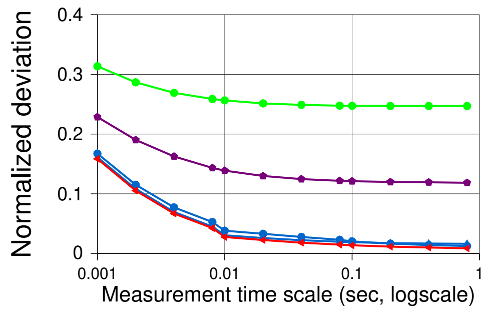
The exact pattern in which the packets are reordered by this method varies with the timing that packets arrive at, and



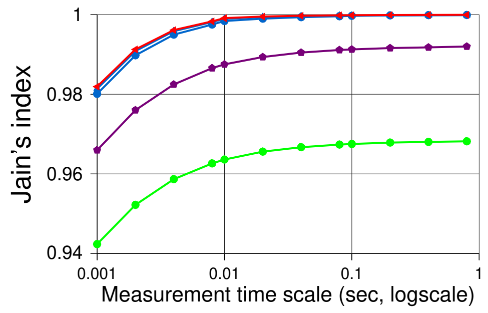
(a) Mean absolute deviation



(b) 99th absolute deviation



(c) Mean normalized deviation



(d) Mean Jain's fairness index

Figure 7: Comparison of load balancing schemes with goodness metrics across all measurement time scales. *HH* denotes inclusion of a heavy-hitter table. Simulation on CAIDA trace augmented with synthetic heavy-hitters

are processed by, the middle-box's kernel. From examination of the netem code, the packets are time stamped when they begin their processing in the kernel. These time stamps are maintained with a resolution of 64 nanoseconds. The latency of 0.9 msec or 1.1 msec is added to the packet's arrival time to get its scheduled departure time. The packet is then placed in an output queue for the target network interface, inserted at the appropriate place so that the queue is maintained in order of scheduled departure time.

The typical pattern of reordering seen during experiments that achieve high throughput is the same as if N consecutive in-order packets arrive at the middle-box and are buffered, then the next M packets are allowed through, passing the N buffered packets on their way to the receiver. Then the N buffered packets are forwarded. For example, with $N = 9$ and $M = 7$, if the sender sent packets D_1 through D_{19} in that order, the packets would arrive at the receiver in the order $D_1, D_{11}, D_{12}, ...D_{17}, D_2, D_3, ...D_{10}, D_{18}, D_{19}, ...$, and would then be in order until the next time the latency went from high to low. Examples of pairs of value (N, M) observed in actual packet traces recorded at the receiver are $(10, 12)$, $(4, 18)$, $(6, 20)$, and $(17, 3)$.

This pattern of adding latency was done for packets in the forward (i.e. client-to-server) and backward directions, with an independent state machine counting packets in each direction.

Figure 9 shows the throughput achieved by a single TCP connection for a Linux client and server, and for a Windows Server 2008 client and server. The throughput measurement was made using *iperf* with data transmission only in the client-to-server direction, and is the average rate over 10 seconds. Each data point plotted is the median throughput of 11 runs with the same conditions. We also repeat the Linux experiments with two other pairs of latency values, $[0.5, 1.5]$ msec (i.e. latency drops by 1.0 msec when it decreases) and $[0.25, 1.75]$ msec (i.e. latency drops by 1.5 msec). With wider latency difference range, we expect worse through-

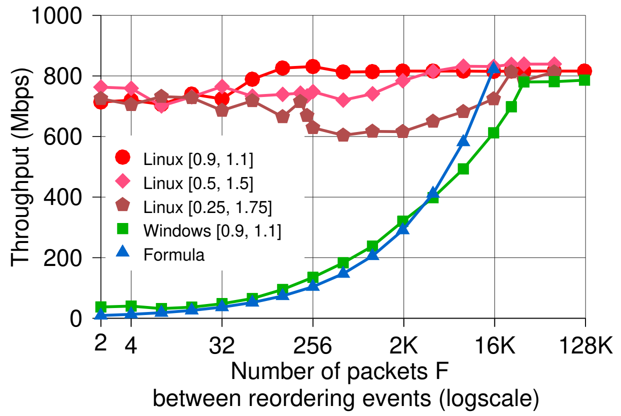


Figure 9: TCP throughput experiments at 1 Gbps. Latency differentiation values in msec.

put because there is more reordering that can be introduced when the drop from high latency to low latency is by a larger quantity of latency. Figure 9 shows that the Linux throughput is still high in all cases, especially when F is no less than about 128.

The "Formula" curve is the TCP throughput predicted by the $1/\sqrt{p}$ model of TCP performance [10]. It is the value of $(MSS \cdot C)/(RTT\sqrt{p})$ for $p = 1/F$, $MSS = 1460$ bytes, $RTT = 2.2$ msec, and $C = 1.22$.

Linux achieves remarkably good throughput even with very frequent reordering. Wu et al [19] ran similar experiments with a middle-box that added normally-distributed random latencies to each forwarded packet. They found similarly good throughput, as long as the standard deviation of the normal distribution was small compared to the mean. They attribute this resilience against reordering to: "an adaptive TCP reordering threshold mechanism. Under Linux, *dupthresh* is adaptively adjusted in the sender to reduce unnecessary retransmissions and spurious congestion window reductions. Some network stacks [...] still implement a static TCP reordering threshold mechanism with a default *dupthresh* value of 3."

Our experiments confirm this. We ran additional experiments where the middle-box dropped a single packet every F packets, and added 1 msec of latency to all packets (thus no reordering). Under these conditions the throughput graphs for both Windows and Linux were nearly identical to the Windows throughput graph of Figure 9, where reordering but no loss is introduced.

We also recorded packet traces on the sender in a few experiments (not used to create the graphs) and used *tcptrace* [18] to estimate the sender's congestion window. This estimate is called "outstanding data" in tcptrace. It is calculated as the largest data sequence number transmitted by the sender, minus the largest cumulative ACK sequence number it has received so far. These graphs showed the Linux sender's congestion window increasing steadily despite packet reordering events, whereas the Windows sender's congestion window dropped every time it received 3 or more duplicate ACKs in a row. Thus the Windows sender is misinterpreting the kinds of reordering we introduce as packet loss, as was also the case with older versions of Linux (circa kernel version 2.6.14 and earlier).

We also ran Linux client to Windows server experiments, and vice versa. Although not identical, they are substantially similar to the Windows throughput graph presented here. It is not enough that Linux is the sender in order to achieve high throughput during reordering. The receiver TCP implementation is also important.

The results here make a strong case that for Linux-to-Linux TCP traffic, per-packet load balancing such as DRR may be acceptable to increase overall application throughput, if the gain in throughput from the more even load balancing is greater than the small throughput reduction caused by packet reordering. For the common case where multiple TCP connections share the network capacity, their competition for bandwidth is likely to be the limiting factor before packet reordering effects are noticeable. For TCP traffic that is not Linux-to-Linux, reordering that causes the sender to react as if there were a packet loss (i.e. 3 or more duplicate ACKs in a row) as often as once every 1024 packets cuts throughput by a factor of 4, according to the results in Figure 9.

Results from the 10 Gbps test-bed:

NetBump [14] allows modification to packets between C and S at 10 Gbps by software (e.g. changing packet headers, adding delay, dropping packets, and crafting packet reordering). We set up a NetBump test-bed with fast machines equipped with Myricom 10 GigE so that packet processing can be done in real time at full 10 Gbps by NetBump techniques to offload work on multiple CPU cores [14]. Since our NetBump test-bed is only Linux-based , we show only the result for Linux TCP in this part.

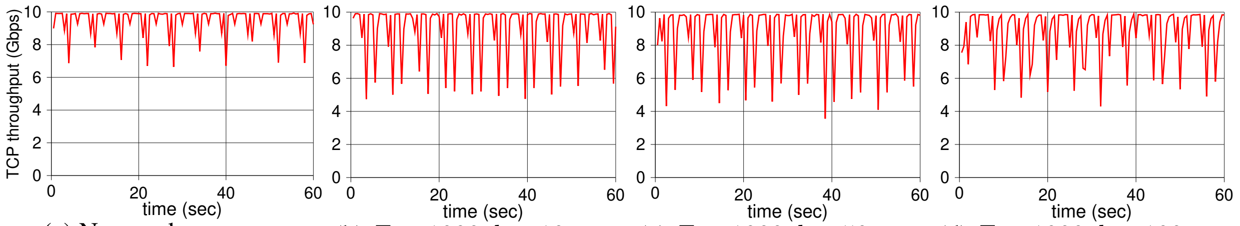
We consider a common pattern of packet reordering caused by load balancing in data center networks. As an example, assume packets D_1 through D_5 go on path P_1 and then packets D_6 onward switch to path P_2 with *lower* latency. Thus although D_1 through D_5 have left the switch, packet D_6 may overtake some of them. A possible scenario at the receiver (in terms of received packets, assume no loss) is $\mathrm{D}_1, \mathrm{D}_2, \mathbf{D_6}, \mathrm{D}_3, \mathbf{D_7}, \mathrm{D}_4, \mathbf{D_8}, \mathrm{D}_5, \mathbf{D_9}, \mathbf{D_{10}}, \mathbf{D_{11}}, \ldots$ back to normal. In other words, it is not a complete burst of packets that are delayed but the delayed burst *interleaves* with the burst switched onto the lower latency path.

We emulate this reordering situation according to the following two parameters.

- Reordering frequency F: we do packet reordering once every F dispatched packets.
- Interleaving burst b: the amount of interleaved packets ($b = 3$ in our example)

Our method to craft the interleaving burst is by buffering upto b packets and alternating them with the subsequent packets accordingly. To avoid infinite packet delay (especially for SYN packets), we hold each packet in the interleaving buffer for at most 1 ms.

Figure 10 plots our TCP throughput benchmark with the *iperf* tool for one minute. Note that we use the default TCP implementation of the Linux operating system installed on the servers (64-bit Debian on Linux kernel 2.6.32). The bandwidth measurement granularity is 0.5 sec. In our test-bed, the round-trip-time (RTT) between the client and server varies in the range $0.2 - 0.5$ ms. From Figure 10, we conclude that the TCP stack on our Linux kernel is highly tolerant to packet reordering. Indeed, we also get consistent



(a) No reorder (avg 9.51 Gbps) (b) $F = 1000, b = 10$ (avg 9.00 Gbps) (c) $F = 1000, b = 50$ (avg 8.82 Gbps) (d) $F = 1000, b = 100$ (avg 8.81 Gbps)

Figure 10: Throughput experiments with one TCP flow at 10 Gbps with interleaving reordering burst by load balancing. F is reordering frequency and b is length of interleaving burst.

results with Figure 10 for several other patterns such as interleaving of short bursts (instead of packets) and per-packet delay differentiation (not shown due to space limit).

6. CONCLUSIONS

Our test-bed results surprised us. They suggest that rather than deploy new transport protocols such as Multipath TCP [15], TCP modifications to recent Linux stacks may allow packet-by-packet rebalancing with only nominal performance loss. This is a bold claim and must be investigated. Do these mechanisms have other side-effects that arise in some scenarios? Is there a fundamental reason why Windows stacks cannot be upgraded with the same mechanisms?

In the interim, at least for Windows machines that are very common, the situation is neatly reversed. We cannot afford to rebalance more than once every $32,000$ packets. Given the uncertainty, load balancing chips today would be wise to have a controllable parameter F. We assert that hardware load balancing such as shown in Figure 4 will be crucial. Software load balancing such as [2] (where the optimal flow assignments for heavy flows is computed by software) will be too slow to allow values of F below 1000 and hence miss balancing opportunities in the future. The Flame hardware described in Figure 4 can also be deployed one router at a time compared to the deployment issues for Hedera [2].

Unlike Hedera, Flame also attempts to initially do a good flow assignment by stealing the basic "place new flow on least loaded link" paradigm from Flare. However, Flame goes beyond Flare by having a more robust link bandwidth estimator (DRE), a more resilient and memory-efficient method to remember flow state by memorizing one of multiple hash functions, and by integrating periodic rebalancing in hardware. In conclusion, while Flame is deeply influenced by Hedera and Flare, we believe it adds significant new mechanisms (summarized in Figure 4) that will be essential for *deployable* and *robust* data center routers at 10 Gbps and beyond. While our paper appears to be narrowly about "load balancing", the broader issue at stake is cheaply providing bandwidth for cluster computation in data centers, which in turn underlies the promise and effectiveness of cloud computing.

7. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2010.

[2] Al-Fares et. al. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[3] Alizadehzy et. al. . Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[4] C. Guo et. al. Bcube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[5] CAIDA. CoralReef Software Suite. `http://www.caida.org/tools/measurement/coralreef/`.

[6] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *SIGCOMM*, 2002.

[7] Greenberg et. al. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[8] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Flare: Responsive Load Balancing Without Packet Reordering. In *ACM CCR*, 2007.

[9] M. Laor and L. Gendel. The Effect of Packet Reordering in a Backbone Link on Application Throughput. In *IEEE Network*, 2002.

[10] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. In *SIGCOMM CCR*, 1997.

[11] McKeown et. al. OpenFlow: Enabling Innovation in College Networks. In *White paper*, 2008.

[12] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. In *PhD thesis*, 1996.

[13] netem. The Linux Foundation. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[14] Porter et. al. User-extensible Active Queue Management with Bumps on the Wire. In *UCSD Tech Report*, 2011.

[15] Raiciu et. al. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.

[16] C. Shannon, E. Aben, kc claffy, and D. Andersen. The CAIDA Anonymized 2008 Internet Traces. `http://www.caida.org/data/passive/passive_2008_dataset.xml`.

[17] tcn. Trace Control for Netem. `http://tcn.hypert.net`.

[18] tcptrace. . `http://www.tcptrace.org`.

[19] W. Wu, P. Demar, and M. Crawford. Sorting Reordered Packets with Interrupt Coalescing. In *Comput. Netw.*, 2009.