Hashing-Based Traffic Splitting Algorithms for Internet Load Balancing

Zhiruo Cao [†] [*] *Zheng Wang* [‡] *Ellen Zegura* [†]

[†] College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{zhiruo,ewz}@cc.gatech.edu

[‡] Bell Labs
Lucent Technologies
Holmdel, NJ 07733
{zhwang}@dnrc.bell-labs.com

GIT–CC–99/14

May, 1999

Abstract

Replication of resources is a key technique for improving Internet performance. Effective use of replicated resources requires good load distribution schemes. We study the performance of several hashing schemes for distributing traffic while preserving the order of packets within a flow. Traffic distribution with per-flow ordering has applications that include balancing traffic across multiple Internet access links and balancing HTTP request load in a web server farm. While hashing schemes for load balancing have been proposed in the past, this is the first comprehensive study of performance using real traffic traces.

We evaluate five direct hashing methods and one table-based hashing method. We find that hashing using a 16-bit CRC over the TCP five-tuple gives excellent load balancing performance. Further, load-adaptive table-based hashing using the exclusive OR of the source and destination IP addresses achieves comparable performance to the 16-bit CRC. Table-based hashing can also distribute load according to unequal weights. We also report on four other schemes with poor to moderate performance.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

[*]The main part of his work was done at Bell Labs, Lucent Technologies

1 Introduction

Load balancing (also known as load sharing) is a key technique for improving the performance and scalability of the Internet. For example, many large enterprise networks are connected to multiple Internet Service Providers (ISPs) to provide redundant connectivity, allow traffic distribution and reduce congestion. A common method for scaling web servers is to operate a farm of machines to which client requests are transparently directed. To achieve high availability, many of the Internet backbones are engineered to have multiple parallel trunks between major Points of Presence. Load balancing is also present within routers, in the form of multiple parallel packet processing engines that eliminate the bottleneck of a single central engine.

For all of these examples, effective use of the replicated resources requires good load balancing schemes. In addition, since the majority of the traffic on the Internet is TCP-based, load balancing schemes need to avoid packet mis-ordering within a TCP flow which can falsely trigger congestion control mechanisms and cause unnecessary throughput degradation.

In this paper, we evaluate a class of hashing-based traffic splitting algorithms which preserve per-flow packet ordering. We consider five hash functions that are "direct," meaning that the hash function produces a value in the range $0 \ldots N-1$, where N is the number of outgoing links. We also consider a table-based generalization that involves hashing to $M \geq N$ bins, then assigning the M bins to the N outgoing links. Table-based hashing requires more state than direct hashing, but has the flexibility to support unequal load distribution and dynamic adaptation.

Our results are obtained by simulating the performance of a traffic splitter, using packet traces taken from two trunks at a major Internet backbone provider. We find that direct hashing using simple functions of the five-tuple (source IP address, destination IP address, source port, destination port and protocol number) results in poor to moderate performance. For example, hashing the destination IP address causes significant imbalance across two links. Using the Internet checksum or the exclusive OR of both the source IP address and destination IP address improves the performance considerably, though moderate imbalance persists. The more computationally complex 16-bit CRC of the five-tuple gives excellent load balancing performance, keeping the load and queue lengths very similar on two links. Equally good load balancing can be achieved using table-based hashing with adaptation, which requires less computation than the CRC but necessitates monitoring the link loads and storing (and adjusting) a mapping from table bins to links.

Table-based hashing has the additional advantage that it can distribute the load according to unequal weights. Further, an index-based version of this scheme can alter the weight distribution with minimal disruption to existing flows. Our results confirm that index-based hashing can accurately achieve a 2:1 weighted distribution when adaptation is also used.

The rest of this paper is organized as follows. In Section 2 we discuss related work in traffic splitting and load balancing. Section 3 describes the behavior of an ideal traffic splitter, explains the requirements for a practical system, and defines the performance metrics that will be used to assess various hashing-based schemes. The

set of schemes that we consider are described in Section 4. The results of our study are described in Section 5, and include analysis of the randomness inherent in the trace data (Section 5.1). We conclude and mention areas for future work in Section 6.

2 Related Work

A simple form of load balancing can be achieved using round robin distribution of packets [4]. This technique has two well-recognized problems in the context of Internet traffic distribution: variable-length packets result in unequal load distribution, and per-packet forwarding does not preserve per-flow FIFO delivery. To overcome the unequal load distribution problem, Adiseshu et al. [2] developed a key insight relating load balancing to fair queueing. Specifically, fair queueing algorithms can be transformed into fair load sharing algorithms. Unfortunately, these algorithms do not preserve per-flow FIFO delivery. FIFO delivery can be guaranteed with the addition of sequence numbers to each packet. Alternatively, a form of quasi-FIFO delivery can be achieved with some complexity at the receiver [2].

Our work differs from this work in two important dimensions. First, we are interested in preserving per-flow FIFO ordering without the addition of sequence numbers to the packets or state at the receiver. Adding an extra header is impractical for IP; keeping state at two ends of a channel works best over point-to-point links, and thus is not particularly well suited for Internet load balancing. Second, we focus on load balancing performance for typical traffic (as exemplified by traces taken from a backbone network), rather than worst case traffic. It is easy to construct pathological cases in which the schemes we consider will perform poorly. However, we argue that if such cases were to arise in practice, they would be symptomatic of far greater problems than load imbalance.

Hashing based on the flow identifier provides a class of methods for distributing load while maintaining per-flow order. Hashing has been widely used in indexing and searching [12]. In the networking context, hashing-based algorithms for address lookup [11], flow identification [5] and packet demultiplexing [7] have been proposed in the past.

The use of hashing for network load balancing is not new. Some commercial router products have implemented simple hashing over the IP destination address to distribute traffic [6]. The OSPF routing protocol supports multiple equal-cost paths [13], but does not specify how such paths should be used. In the OSPF Optimized Multipath protocol (OSPF-OMP) [17], a number of possible approaches for load balancing over multiple paths are mentioned, including per-packet round robin, dividing destination prefixes among available next hops in the forwarding table, and dividing traffic according to a hash function applied to the source and destination pair. The CRC16 algorithm is proposed as the hashing function. The hash output is then compared to the boundary value for each next hop to determine the outgoing link. The performance of the proposed schemes is not evaluated with simulation or real network measurement. In the simulation comparing OSPF-OMP and a direct connection, perfect hashing performance is assumed [16].

A traffic splitting scheme using random numbers is proposed in [14]. It applies the name-based mappings approach to load balancing [15]. In this scheme, each next-hop

is assigned with a weight based on a simple pseudo-random number function seeded with the flow identifier and the next-hop identifier. When a packet arrives, the weights are generated, and the next-hop receiving the highest weight is used for forwarding. The scheme is approximately N times as expensive as a hashing-based scheme, where N is the number of outgoing links. No performance study on the proposed scheme is presented.

It is clear that although hashing-based schemes for traffic splitting have been proposed for in the past, and some simple schemes have even been implemented in commercial products, the performance of such schemes has not been adequately evaluated. This is the first comprehensive performance study on a wide range of hashing-based schemes, using real packet traces from backbone networks.

We end this section by mentioning that load balancing has also been widely studied and deployed in telecommunication networks [8], to allow providers to offer wideband channels by combining multiple narrowband trunks. Inverse multiplexers operating on 56kbps and 64kbps circuit switched channels are commercially available. The standardization of inverse multiplexers is coordinated by the BONDING consortium [9]. However, the techniques used in inverse multiplexing are typically applicable over point-to-point links, but not to general Internet load balancing.

3 Framework

In this section, we describe the behavior of an ideal traffic splitter, explain the requirements for a practical system, and define the performance metrics for assessing various schemes.

3.1 Reference Model

A load balancing system typically comprises a traffic splitter and multiple outgoing links as shown in Figure 1. In such a system, the traffic splitter receives an incoming packet from a higher-speed link and forwards it to one of the lower-speed outgoing links. A good load balancing system should be able to split the traffic to the multiple outgoing links evenly or by some pre-defined proportion.

In [2], it has been observed that there is a close relationship between fair queuing and load balancing. We now extend their observation to a mathematical model.

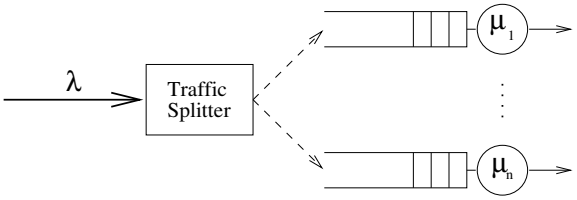


Figure 1: Reference Model

Let us first look at an ideal fluid model where the traffic is infinitely divisible. Suppose that there are N outgoing links in the load balancing system, and the capacity of link

i is μ_i. Let $S_i(\tau, t)$ be the amount of traffic forwarded to link i during the period $[\tau, t]$. The ideal load balancing system should perform as well as the corresponding system with a single outgoing link of capacity $\sum \mu_i$. Therefore, the ideal system should satisfy the following for any period $[\tau, t]$:

$$\frac{S_i(\tau, t)}{S_j(\tau, t)} = \frac{\mu_i}{\mu_j} \tag{1}$$

The traffic load is essentially split in proportion to the rates of the outgoing links. At any time instance, the traffic load is perfectly balanced; all outgoing links are busy or idle at the same time. Such a system is work-conserving; there is no bandwidth lost because of load balancing.

Ideal load balancing cannot be achieved in a real network system. As the basic unit of forwarding is at least a single packet, a packetized load balancing system is no longer work-conserving. For example, suppose that a load balancing system has two outgoing links of the same capacity. Assume that the system is initially idle, then a single packet arrives. The packet is forwarded to one of the two outgoing link. Note that the packet is serviced with half of the total bandwidth available, thus it will take twice the amount of time to transmit compared with an ideal system. During this period, one of two outgoing links is busy servicing the packet while the other link remains idle. In a practical system, the traffic splitter may send several packets in a row to the same outgoing link, and thus increase the waste of bandwidth.

In a packetized system, consider the worst case that all outgoing links have been idle since time τ when a packet of maximum size P_{max} arrives and no more packets are coming until the packet is served. Assume the packet is forwarded onto link i. During the service period, Equation 1 no longer holds because $\frac{S_i(\tau,t)}{\mu_i} - \frac{S_j(\tau,t)}{\mu_j} = \frac{C}{\mu_i} > 0$, where C is a fraction of the packet that has been serviced during the period. Therefore, in a packetized system, the ideal load balancing should satisfy the following:

$$|\frac{S_i(\tau, t)}{\mu_i} - \frac{S_j(\tau, t)}{\mu_j}| \leq \frac{P_{max}}{min(\mu_i, \mu_j)} \tag{2}$$

over any interval $[\tau, t]$, where P_{max} is the maximum size of packet. That is, the difference between the time link i is busy and the time link j is busy should be no more than the time to send a largest packet over the slower link.

3.2 Requirements

There are a number of basic requirements that traffic splitting schemes for Internet load balancing should meet:

- *Low Overhead.* Traffic splitting is part of the packet forwarding path that is executed for every packet, thus the per-packet overhead it introduces is a major consideration. Traffic splitting algorithms need to be very simple so that the computation can be done within the limited cycles that a router has to forward a packet. They should also keep no or little state and be easy to implement in hardware.

- *High Efficiency.* The efficiency of a load balancing system largely depends on how well the traffic splitting is done. Poor traffic distribution will result in uneven link utilization and loss of bandwidth. A traffic splitter should try to distribute traffic as close as possible to the reference model.

- *Per-Flow Ordering.* The majority of the Internet traffic is carried by TCP. For TCP, packet mis-ordering within a TCP flow can produce a false congestion signal and cause unnecessary throughput degradation. It is, therefore, an essential requirement that the traffic splitting algorithms maintain ordering within a TCP flow. Since a TCP flow is identified by the five-tuple in the packet header (Source IP Address, Destination IP Address, Source Port, Destination Port, Protocol Number, often referred as the 5-tuple), a traffic splitting algorithm that uses anything other than a subset of the 5-tuple might cause mis-ordering unless additional mechanisms are used.

Let us now apply the above requirements to some of the possible traffic splitting approaches. Take packet-by-packet round robin or some form of fair queuing for example. The overheads are low and the performance is typically close to optimal. However, per-flow ordering cannot be guaranteed unless additional mechanisms, such as sequence numbers or state keeping, are added. Such additional mechanisms will increase the overhead drastically, and in many cases, only work over point-to-point links.

Hashing-based traffic splitting algorithms are stateless. Most hash functions are fairly easy to compute, particularly with hardware assistance. What is more, if the hash functions use any combination of the five-tuple as input, per-flow ordering can be preserved[1]. As we will show later in this paper, many of the hashing-based schemes perform fairly well. Thus, hashing-based schemes meet the above requirements quite well and offer the best tradeoffs.

3.3 Performance Metrics

We now discuss the basic performance metrics for evaluating traffic splitting algorithms for Internet load balancing.

- *Load Distribution.* An obvious performance metric is the distribution of bytes over time among the multiple outgoing links, since this is the basic objective of load balancing. As we have discussed at the beginning of this section, in an ideal system, the traffic load is distributed in proportion to the rates of the outgoing links.

- *Queue Length.* The load distribution curve usually fluctuates rather than remains constant over the time. To some extent, such fluctuation of load can be absolved by buffering. Thus, the cumulative effects of load balancing are better reflected by the dynamics of queue length of outgoing links. In our analysis, we also use the queue length for each load-balanced link as a performance metric. An advantage of the queue length metric is that it takes into account the fact that load distribution discrepancy during a lightly loaded period has far less real effect than a heavily

[1] This is true because all packets within the same TCP flow have the same five-tuple, thus the output of the hash function with the five-tuple as input should always be the same.

loaded period. A good traffic splitting algorithm may not necessarily have perfect load distribution at all time instances, but it should be able to keep the backlogs of multiple load-balanced links small and balanced.

- *Non-Work-Conserving Idle Time.* As we have discussed earlier, a packetized load balancing system is non-work-conserving. We define the length of the period when at least one link is idle while others are busy as the non-work-conserving idle time (or idle time for short). The idle time metric captures the non-work-conserving inclination of the system: the larger the idle time metric is, the farther away the system skews from work-conserving, and hence the less efficient the load balancing is.

4 Hashing-Based Approaches

In this section, we describe a number of possible hashing-based schemes for load balancing that we will evaluate in the next section.

4.1 Direct Hashing

Direct hashing is a simple form of hashing-based traffic splitting. With direct hashing, the traffic splitter applies a hash function with a set of fields of the five-tuple, and uses the hash value to select the outgoing link. Direct hashing is very simple to implement and requires no extra state to be maintained. In this paper, we consider the following five direct hashing schemes.

4.1.1 Hashing of Destination Address

The simplest scheme is to hash the IP destination address modulo the number of outgoing links N. It can be expressed as:

$$H(\cdot) = DestIP \ mod \ N$$

In this scheme, if $N = 2^k$, we effectively use the last few bits of the destination address as an index of the outgoing link.

4.1.2 Hashing Using XOR Folding of Destination Address

XOR folding has been used in many hash functions, and has been shown to provide good performance in other applications [11]. We propose a hash function with XOR folding of the destination IP address. This hash function can be expressed as:

$$H(\cdot) = (D_1 \oplus D_2 \oplus D_3 \oplus D_4) \ mod \ N$$

where D_i is the ith octet of the destination IP address. This approach utilizes more bits of the destination address in selecting the link.

4.1.3 Hashing Using XOR Folding of Source and Destination Addresses

A simple modification to the previous hash function is to include the source address in the computation, i.e., XOR folding with both the destination IP address and the source IP address. This hash function can be expressed as:

$$H(\cdot) = (S_1 \oplus S_2 \oplus S_3 \oplus S_4 \oplus D_1 \oplus D_2 \oplus D_3 \oplus D_4) \ mod \ N$$

where S_i and D_i are the ith octets of the source and destination IP addresses respectively.

4.1.4 Internet Checksum

The Internet Checksum algorithm proposed in RFC1071 [1, 3] is relatively simple to compute and is also a good hash function. In this paper, we examine its performance as a traffic splitting algorithm. We feed the five-tuple as input to the 16-bit Internet Checksum computation. The index of the outgoing link is calculated from the checksum result modulo by N. This hash function can be expressed as:

$$H(\cdot) = CheckSum(5 - tuple) \ mod \ N$$

4.1.5 CRC16

The 16-bit CRC (Cyclic Redundant Checksum) algorithm [10] has been proposed as a possible candidate for load balancing. Although CRC16 is certainly more complex compared with the other hash functions discussed above, it is easily implemented with hardware assistance.

In the CRC16 scheme, the traffic splitter takes the five-tuple, applies CRC16, and takes modulo by N to obtain the outgoing link. The hash function can be expressed as:

$$H(\cdot) = CRC16(5 - tuple) \ mod \ N$$

4.2 Table-Based Hashing

Direct hashing is simple but has some limitations. First, direct hashing can only split load into equal amounts to multiple outgoing paths. In reality, it is not always desirable to distribute the traffic load evenly. For example, an organization may have links to two Internet backbones with one link twice the speed of the other. The organization may wish to distribute the traffic with a 2:1 ratio. With direct hashing, it is difficult to tune the load distribution; the only way of changing traffic splitting is to use a different hash function. The table-based hashing approach we discuss below addresses both issues by separating traffic splitting and load allocation.

A table-based hashing scheme first splits a traffic stream into M bins. The M bins are then mapped to N outgoing links based on an allocation table (see Figure 2). By changing the allocation of the bins to the outgoing links, one can distribute traffic in a certain defined ratio. One can also fine tune the performance of traffic splitting by simple adjustment to the allocation table. The ratio of M and N determines the granularity

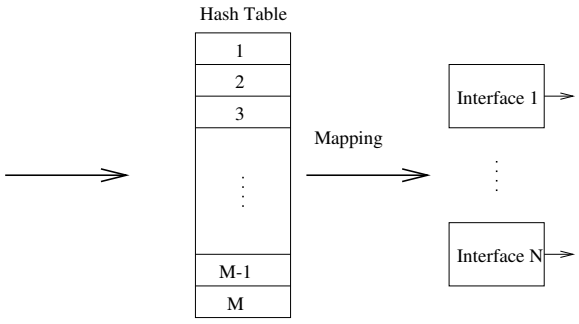


Figure 2: Table-based Hashing

of adjustment. Typically, M is one or two orders of magnitude larger than N, thus one can split the load at a fairly fine granularity. When $M = N$ with one-to-one mapping, table-based hashing becomes direct hashing, hence one can view direct hashing as a special case of the table-based approach.

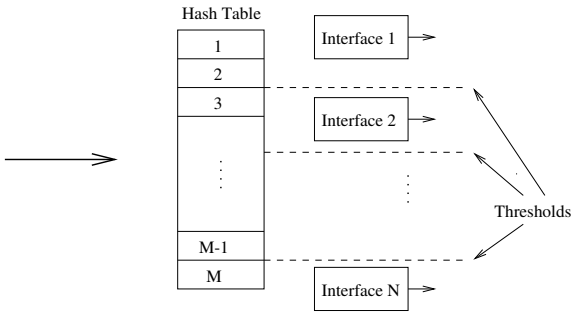


Figure 3: Table-based Hashing with Thresholds Mapping

There are two basic approaches for implementing a table-based scheme. One approach requires $N - 1$ thresholds to be maintained, one for each outgoing link (see Figure 3). The thresholds are used to divide the M bins into N partitions. When a packet arrives, the traffic splitter computes the hashing and compares hash value against the $N - 1$ thresholds to determine the outgoing link. For example, suppose we want to split load over two outgoing links with a 2:1 ratio. We can simply set the threshold to $M/3$. For each arriving packet, we compute the hash value, and then compare with the threshold. If the hash value is greater than $M/3$, the packet is sent to the first link, otherwise to the second link.

A more flexible approach is to associate the outgoing link index with each of the M bins (see Figure 4). This index-based approach requires more memory than the threshold-based approach (M indexes versus $N - 1$ thresholds). On the other hand, the mapping from the hash value to the outgoing link is simpler with the index-based approach. It can be done with a direct table lookup whereas with the threshold-based approach, the hash value has to be compared with the $N - 1$ thresholds.

The index-based approach is more flexible since each of the M bins can be assigned to N outgoing links independently. This can be used to minimize disruption to the

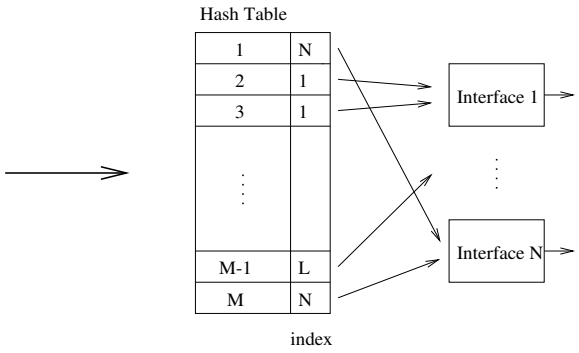


Figure 4: Index-Based Approach to Minimize Flow Disruption

Figure 5: 104-bit 5-Tuple as Flow ID

existing traffic when load splitting is adjusted, or new links are added or shut down. In contrast, the threshold-based approach may cause a significant amount of flows to change their outgoing links. For an example, suppose that a new link is added to the existing two load balanced links. If the load is to be evenly distributed, 1/2 of the traffic flows are re-assigned to different outgoing links. This may potentially cause significant transient packet mis-ordering to the affected flows.

5 Performance Evaluation

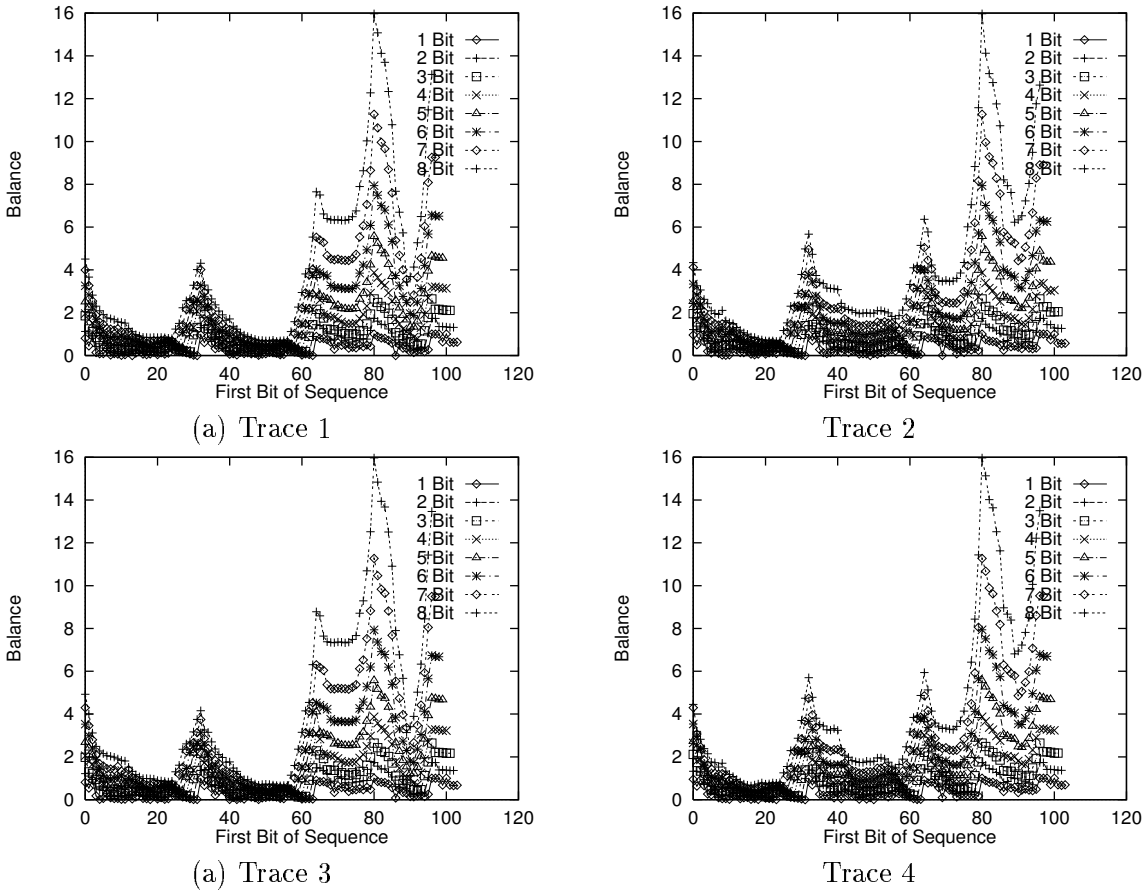
In this section, we present simulation results on the performance of the hashing-based schemes we described in the previous section. We first look at the randomness properties of the traffic traces, and then examine load distribution, queue dynamics and idle time.

5.1 Traffic Traces

The simulation is based on real packet traces collected on MCI Internet Backbone (over one OC12 trunk and one OC3 trunk) during the summer of 1998. The four sets of traces contain complete packet headers and timing information of about 8 million packets. Traces 1 and 3 came from the same trunk, while traces 2 and 4 came from the other trunk.

Since hashing-based approaches for traffic splitting essentially try to exploit the randomness inherent in the traffic streams, the performance of any algorithms is inevitably constrained by the properties of the traffic itself. Thus, we first look at certain properties of the traffic traces and examine how they affect the performance of traffic splitting.

We first consider the randomness of the individual bits in the five-tuple. We construct a 104-bit string with the five-tuple (see Figure 5), and then take m bits starting from the ith bit of the 104-bit string. The m bits are used as an index to sort packets into 2^m bins. The standard deviation of number of packets in each bin indicates



(a) Trace 1 Trace 2

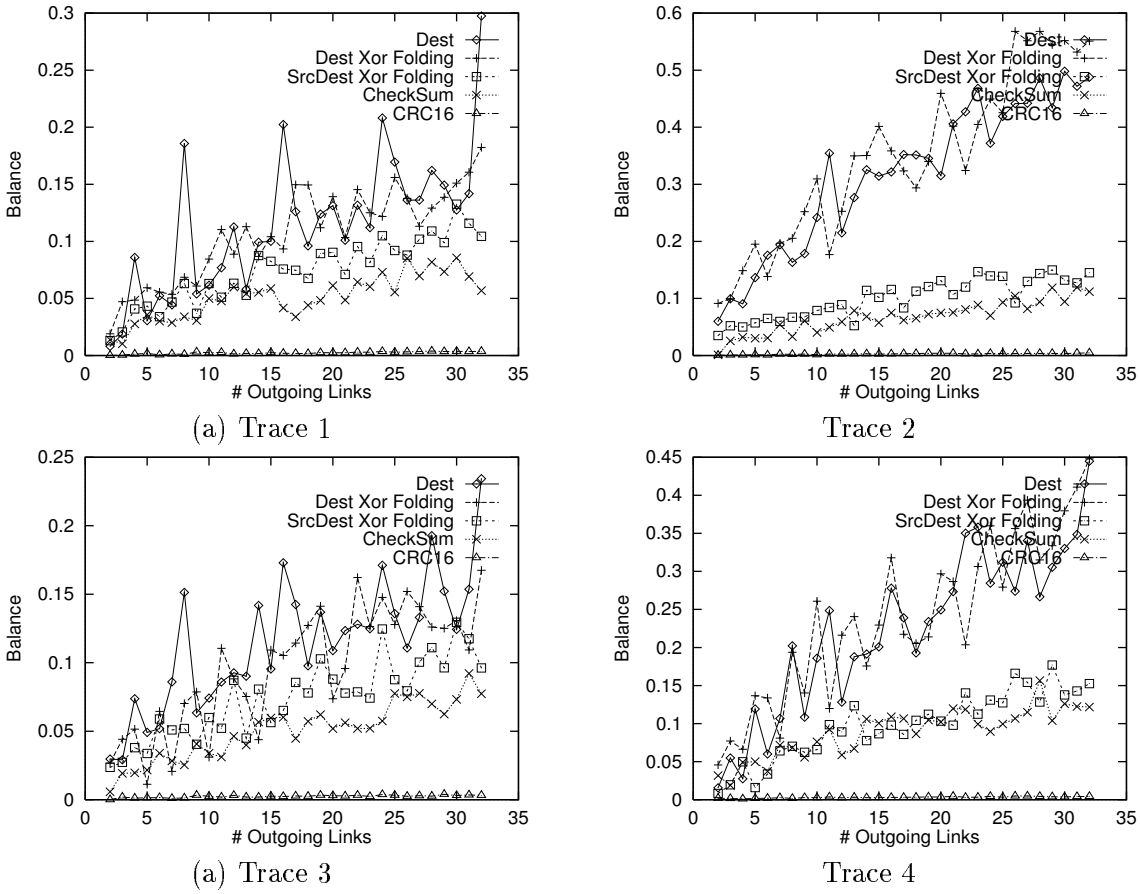
(a) Trace 3 Trace 4

Figure 6: Sorting Packets with M bits from ith bit

the performance of using certain bits for traffic splitting. The results are plotted in Figure 6 for the four traces and m from 1 to 8. The x-axis represents the starting bit position of the m-bit sequence, while the y-axis corresponds to the ratio of the standard deviation to the average. A lower standard deviation to average ratio indicates better performance. The results show that the bits close to the 31th bit and the 63rd bit (the least significant bit of source IP address and the least significant bit of destination IP address respectively) give better hashing performance. This shows that the low order bits in source and destination addresses tend to be more random.

To illustrate the impact of different hash functions, we apply the five different direct hashing schemes described in Section 4.1 to the same 104-bit string. As shown in Figure 7, there are significant differences in performance, and CRC16 performs consistently better than others.

Another two key properties that have significant impact on the performance are the frequency of consecutive packets from the same flow, and the number of flows within a time window. The traffic splitting algorithms we discuss in this paper only use the five-tuple to maintain packet ordering within a flow. Since we cannot split traffic at a sub-flow granularity, consecutive packets from the same flow can be viewed as a



(a) Trace 1 Trace 2

(a) Trace 3 Trace 4

Figure 7: Sorting Packets with Direct Hashing Schemes

single logical packet train that can not be further divided by the traffic splitter. It is obvious that a large number of long packet trains will have negative effects on the traffic splitting performance. Figure 8 shows the statistics of consecutive traffic (in bytes) from the same flow. The typical sizes of the packet trains are around 40, 512 and 1500 bytes, the common sizes of single packets, which indicates that packets from different flows are mixed fairly well. The maximum packet train sizes for the four traces are 4540, 7488, 4136 and 7500 bytes respectively.

The number of flows present in a given time window is also critical to the performance. With a small number of large flows, traffic splitting becomes significantly harder compared with a large number of small flows. Figure 9 shows the number of flows and packets that are observed in a given time window. The middle point of each errorbar shows the average number of flows or packets during each given time period. The upper and lower point of each errorbar show the maximum number and minimum number respectively. It seems that the number of flows in these packet traces is sufficiently large for traffic splitting.

In the analysis so far, we consider the packet traces only as bit strings. While such an analysis can illustrate the randomness properties of the traffic traces and the ability

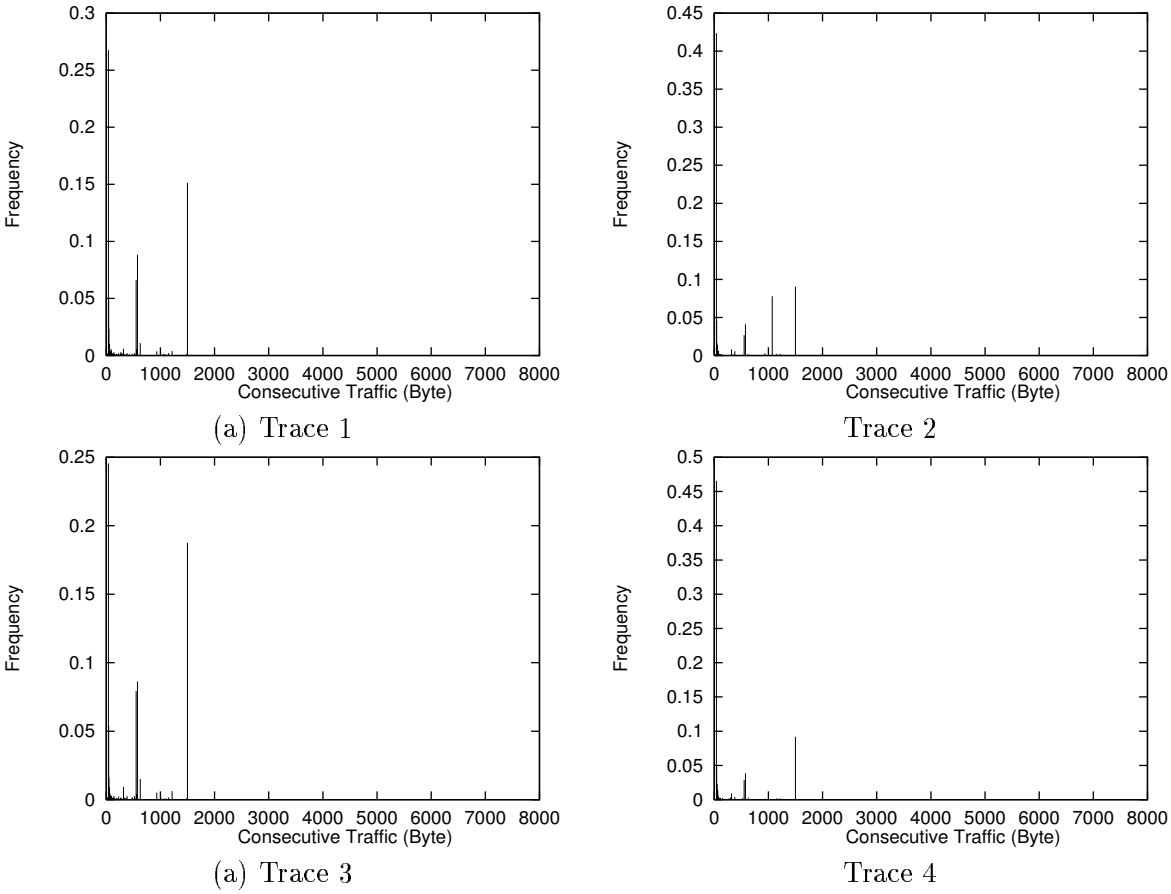


Figure 8: Frequency of Consecutive Traffic (in bytes) from the Same Flow

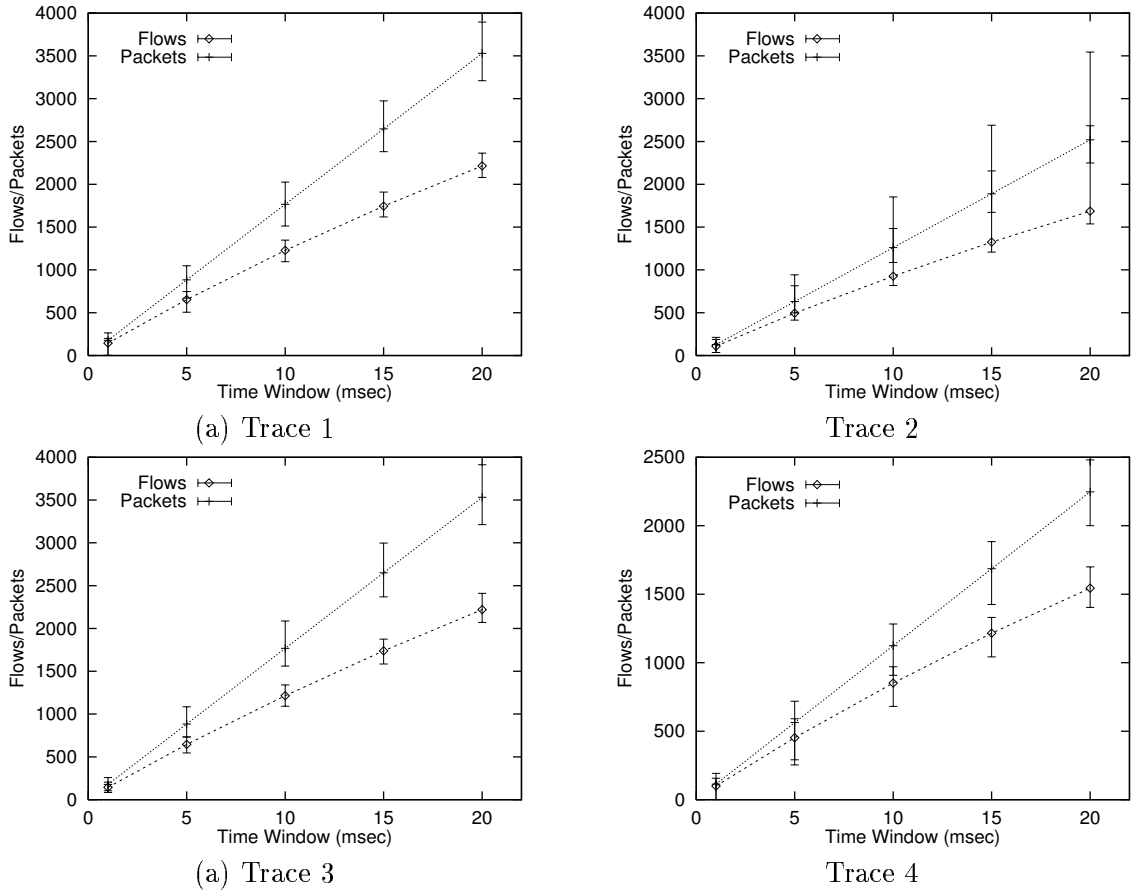
of hashing-based schemes to exploit those properties, it may not necessarily reflect the true performance of traffic splitting in a real network. In Section 5.2, we examine load distribution and queue dynamics by taking into account the timing of packet arrival and the variable packet sizes.

5.2 Load Distribution and Queue Dynamics

The simulation is driven by the traffic traces we discussed in Section 5.1. When a packet arrives, the traffic splitter applies a hash function using some or all of the 5-tuple as the input. The resulting hash value is used to determine the outgoing link. For direct hashing, the hash value is simply the index of the outgoing link. For table-based hashing, the index of the outgoing link is generated by comparing the hash value with thresholds or a further table lookup as we described in Section 4.1.

For simplicity, we focus on the common case of load balancing over two outgoing links of the same capacity. The capacity of outgoing links is set to half of the average rate of the traffic traces. We will also present results for the cases where there are a fairly large number of outgoing links or the two outgoing links have different speeds.

Due to space limitations, in this section we only show the simulation results with



(a) Trace 1 Trace 2

(a) Trace 3 Trace 4

Figure 9: Number of Flows Seen in a Given Time Window

traffic Trace 1. The results for the other three sets of traces are similar. For each hash function we simulated, two graphs are shown side by side, with traffic load for each outgoing link on the left side and the queue length over time on the right side.

5.2.1 Destination IP Address

Figure 10 shows the traffic splitting performance based on hashing of the destination IP address. Since there are only two links, this corresponds to using the low order bit of the destination address to select the outgoing link. As can be seen from the left plot, there exists a significant gap between the traffic loads over the two outgoing links. This difference in load distribution is amplified in the queue length plot on the right. While one queue quickly builds up, the other queue is idle for most of the time. Bandwidth is lost as a result.

5.2.2 Destination IP XOR Folding

Figure 11 shows the traffic splitting performance based on hashing of octet-wise XOR folding of destination IP address. The two figures are very similar to those of hashing

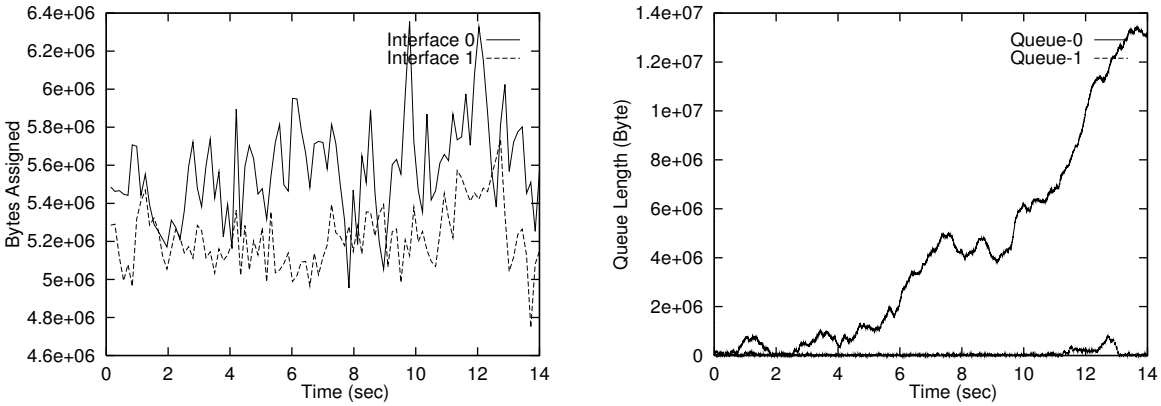


Figure 10: Direct Hashing - Destination IP Address

of destination IP address. It is interesting to observe, however, that outgoing link 0 is overloaded in Figure 10 while the load distribution is reversed in Figure 11.

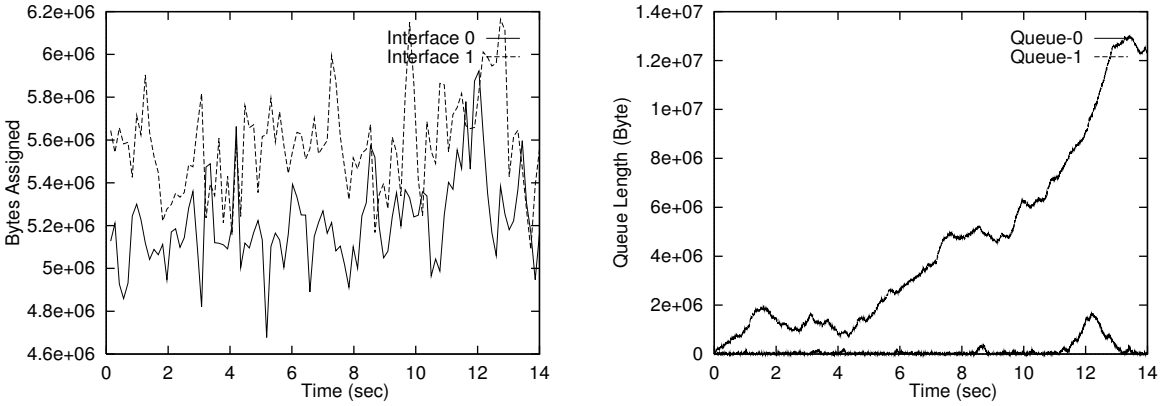


Figure 11: Direct Hashing - XOR Folding of Destination IP Address

5.2.3 XOR Folding of Source and Destination IP Addresses

Figure 12 shows the traffic splitting performance based on hashing of octet-wise XOR folding of both the Source IP address and the destination IP address (See Section 4.1.3). Compared with the previous two results, this one is significantly better both in terms of load distribution and queue length. The result indicates that hashing with both the source address and the destination address can improve the performance over hashing with only the destination IP address.

We believe that part of the reason is the randomness of source IP addresses and destination IP addresses depends on the location in the Internet. For example, packets going to a major web site (e.g., Yahoo) will have similar destination IP addresses and different source IP addresses, and vice versa for packets coming from the web site. Thus, combining both the source address and destination exploits the randomness better.

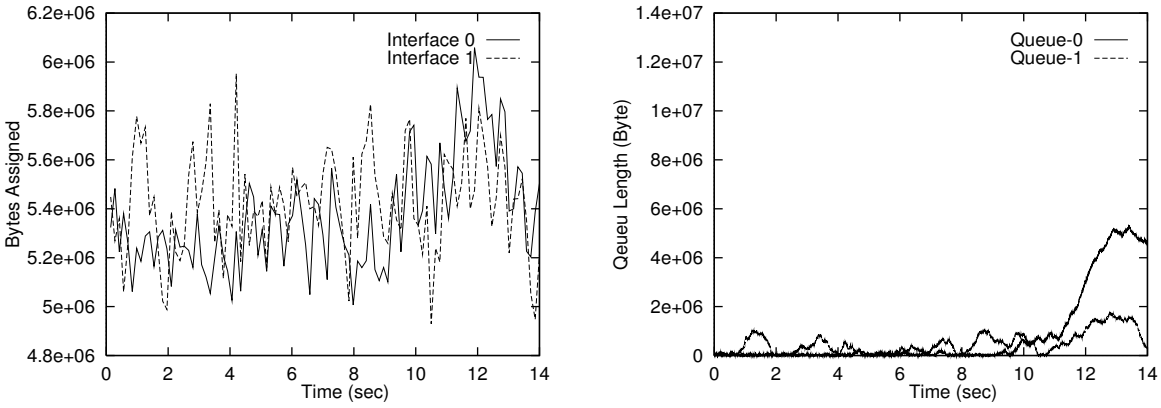


Figure 12: Direct Hashing - XOR Folding of Source and Destination IP Address

5.2.4 Internet Checksum

Figure 13 shows the traffic splitting performance based on the checksum hashing described in Section 4.1.4. The overall performance is close to that of hashing of octet-wise XOR folding of both the source IP address and the destination IP address. Although the load distribution seems to fluctuate more severely compared with Figure 12, the difference in the queue dynamics is minimal.

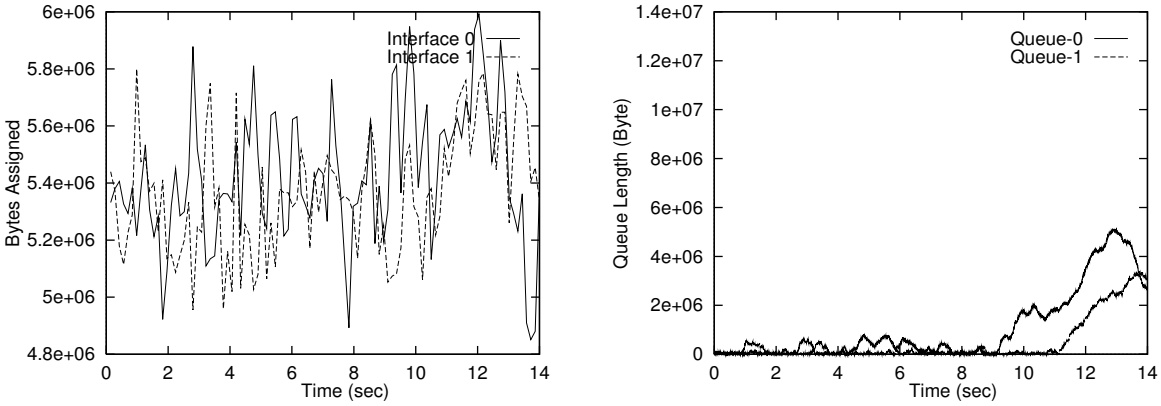


Figure 13: Direct Hashing - CheckSum

5.2.5 CRC16

Figure 14 shows the traffic splitting performance based on CRC16 as described in Section 4.1.5. As can be seen from the left plot, two load curves follow each other fairly nicely. In the queue length plot the right-hand side, the queues associated with outgoing links are relatively small and very well correlated. The performance of CRC16 is certainly much better than the other ones.

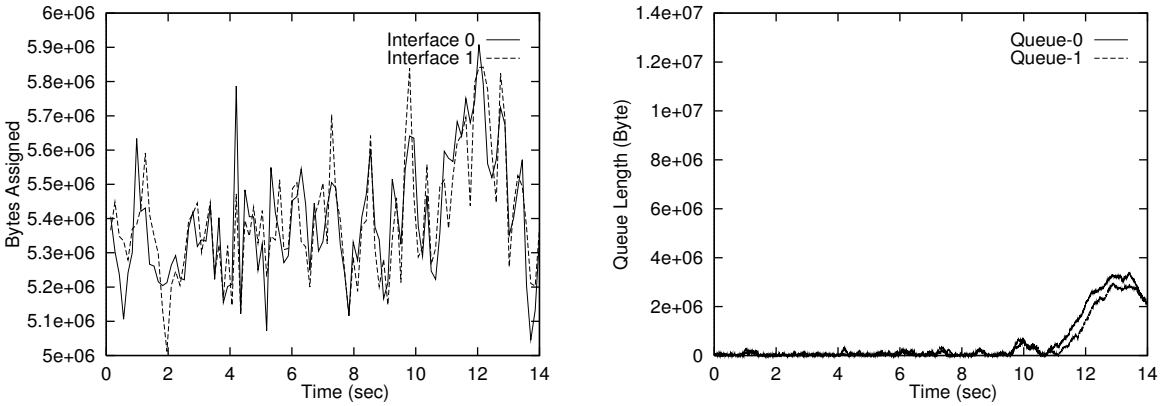


Figure 14: Direct Hashing - CRC16

5.2.6 Table-Based Hashing

Figure 15 shows the traffic splitting performance of table-based hashing as described in Section 4.2. A 16-bit XOR folding function based on source and destination IP addresses is used to hash incoming packets into 64K bins. The packets that fall into the upper 32K bins go to the first outgoing link and the rest to the second outgoing link. Compared with the direct hashing shown in Figure 12, the difference is not significant.

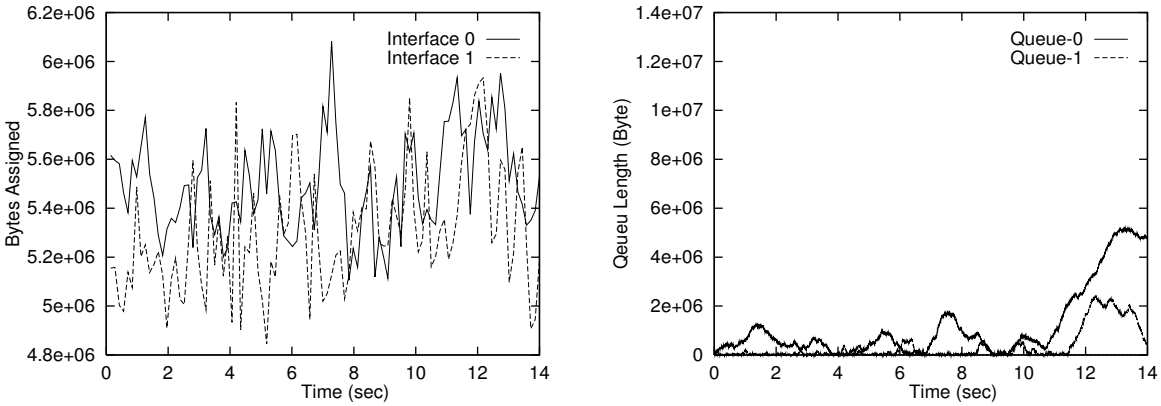


Figure 15: Table-Based Hashing - XOR Folding of Source and Destination IP Addresses

However, a table-based hashing scheme can easily support fine tuning and adaption. A simple approach we consider here is to monitor the load on each outgoing link and to adjust bin assignment periodically. At each adjustment point, we move the bins from heavy loaded links to light loaded links to re-balance the traffic splitting.

We run the same simulation as shown in Figure 15 with adaptation, and the results are shown in Figure 16. As we have expected, the performance is significantly improved through adaptation.

As we discussed in Section 4.2, table-based hashing schemes can split the traffic in some proportion rather than in equal amounts. Figure 17 shows the performance of weighted (2:1) load balancing based on XOR folding of source and destination IP

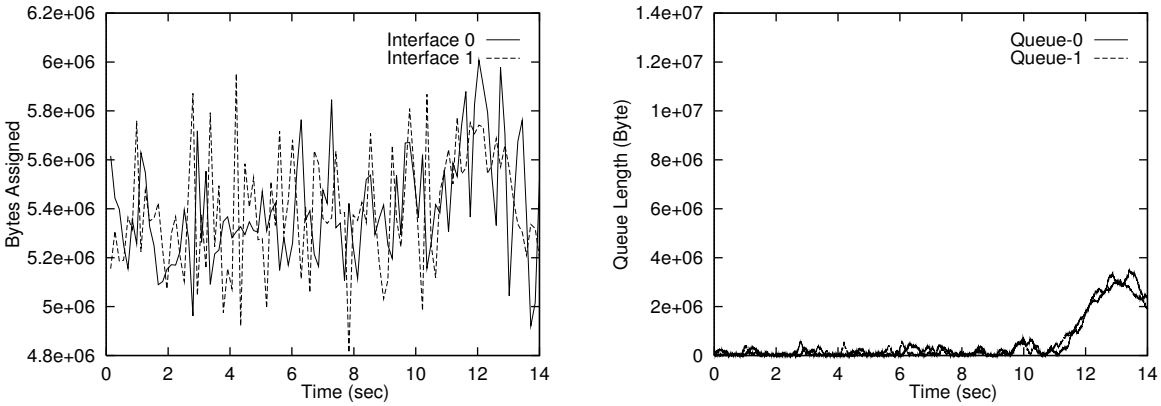


Figure 16: Table-Based Hashing — with Adaptation

addresses. The 2:1 ratio is kept fairly well throughout the period.

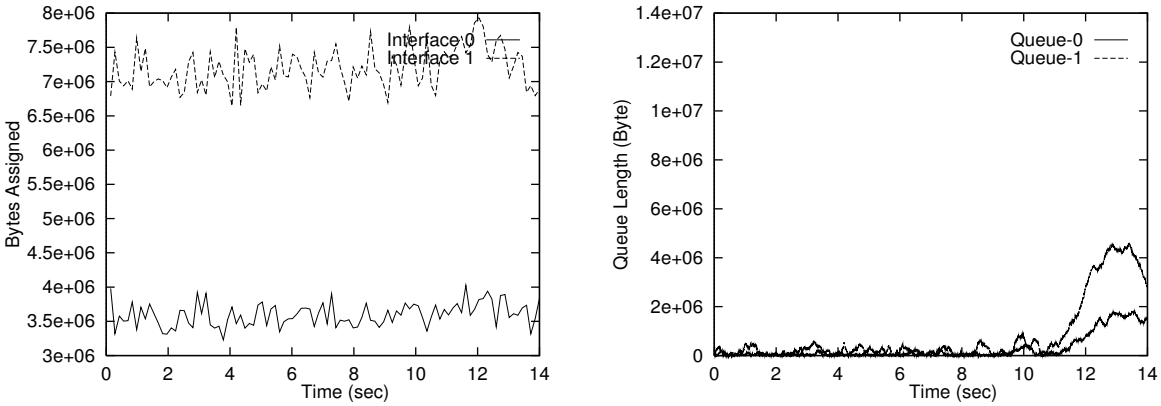


Figure 17: Table-Based Hashing - Weighted Load Balancing with Adaptation

5.3 Simulations of Ten Outgoing Links

We also conducted extensive simulations with more than two outgoing links. Most of the results are fairly consistent with what we showed so far. Due to space limitations, we only select two results here and show them in Figure 18. The plot in the left side of Figure 18 shows the queue performance of direct XOR-folding of source and destination IP addresses over 10 outgoing links, and the plot in the right side shows the queue performance of direct CRC16 hashing over 10 outgoing links. As we can see, CRC16 performs significantly better and more consistently.

5.4 Non-Work-Conserving Idle Time

We now look at the idle time as discussed in Section 3.3 for the five direct hashing schemes. In our simulation, we track the queues for the outgoing links. For any period,

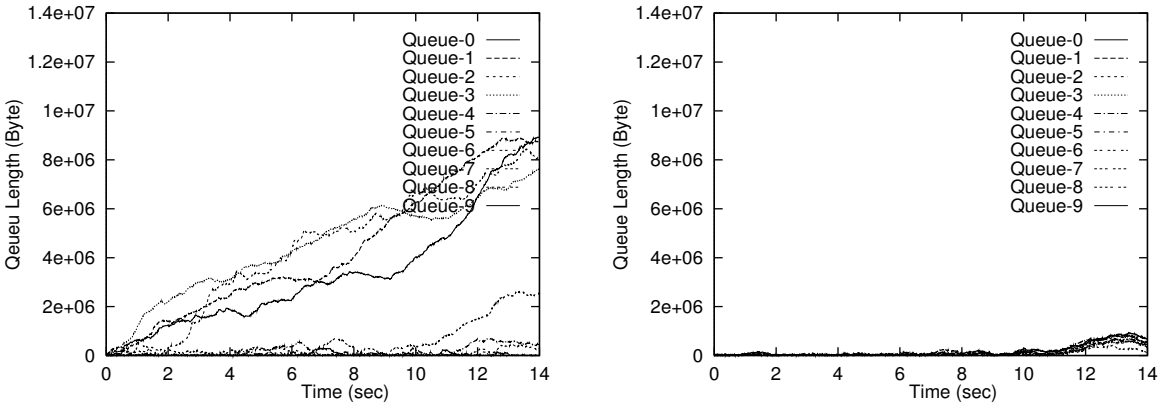


Figure 18: Hashing — Source Destinations Xor Folding and CRC16 with 10 outgoing links

if one queue is busy while the other is idle, we record this period as non-work-conserving idle time. The idle time reflects the resources wasted due to load balancing.

Plots in Figure 19 show the idle time measurement of the load balancing system over the simulation. The Lightest-Loaded First (LLF) is used here as a reference benchmark. LLF always chooses the link that has served the least during the past cycle (which is 0.14s in the simulation), so that it is close optimal under the packetized load balancing model but does not guarantee per-flow ordering. The figures show that CRC16-based hashing consistently performs better than other schemes. The hashing scheme based only on destination IP address performs the worst in most traces.

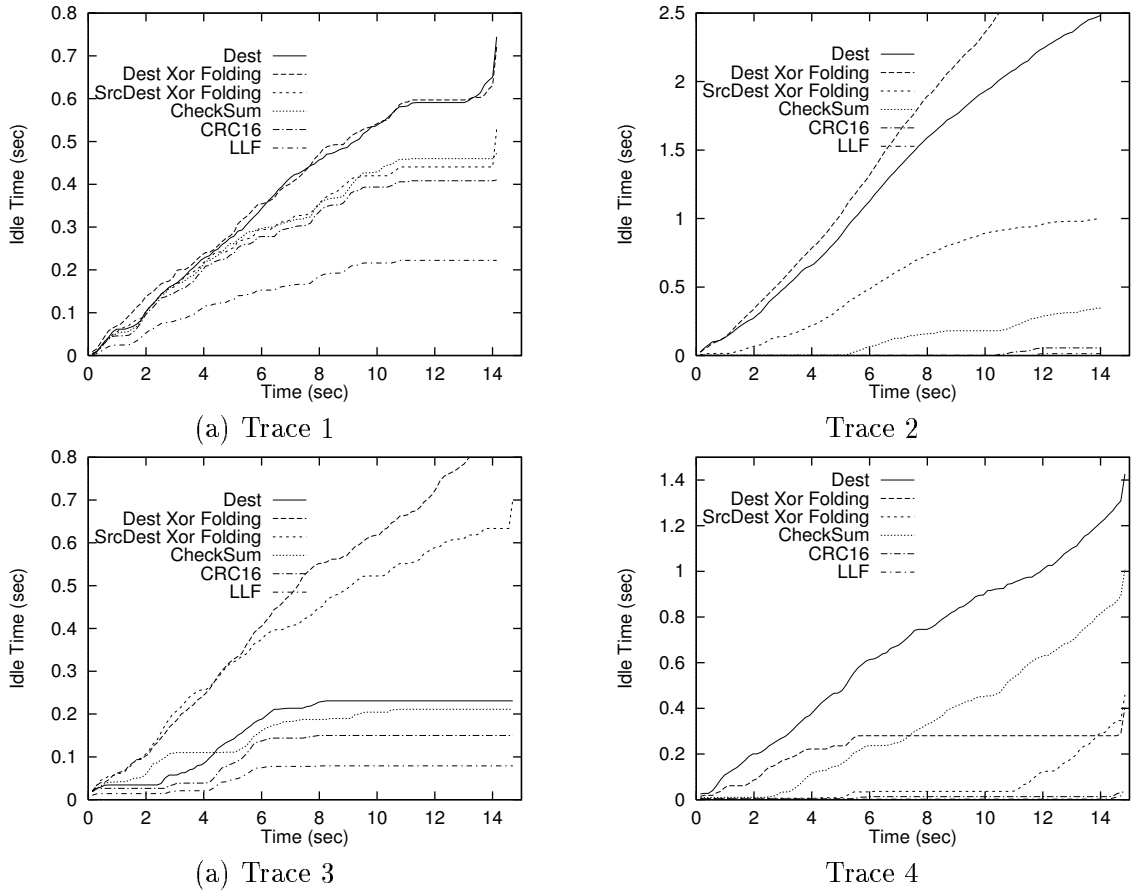
6 Conclusion and Future Work

In this paper, we examined hashing-based traffic splitting algorithms for Internet load balancing. Overall, hashing-based approach meets the requirements fairly well in terms of overheads, performance and maintaining per-flow ordering. The table-based hashing approach can support load balancing with different weights, and fine tuning of traffic splitting. Our simulation results indicate that there are significant differences in performance among various schemes. The CRC16-based scheme has shown to perform very well and consistently better than other schemes.

Obviously our results are based on the traffic traces we have. We plan to repeat our simulation with more traffic traces and in particular traces from different part of the Internet. Another area for splitting future research is to develop new hashing-based algorithms that have less computational complexity than CRC16 but offer similar performance.

Acknowledgment

We would like to thank Kevin Thompson and Greg Miller of MCI for their help with the traffic traces.



(a) Trace 1 Trace 2

(a) Trace 3 Trace 4

Figure 19: Idle Measure

References

[1] Internet Protocol – Specification. *RFC 791*, September 1981.

[2] H. Adiseshu, G. Parulkar, and G. Varghese. A Reliable and Scalable Striping Protocol. In *Proceedings of SIGCOMM 96*, September 1996.

[3] R. Braden, D. Borman, and C. Partridge. Computing the Internet Checksum. *RFC 1071*, September 1988.

[4] C. Brendan, S. Traw, and J. Smith. Striping Within the Network Subsystem. *IEEE Network*, July/August 1995.

[5] G. Chandranmenon and G. Varghese. Trading Packet Headers for Packet Processing. In *Proceedings oGf SIGCOMM 95*, September 1994.

[6] Cisco Systems. Load Balancing with Cisco Express Forwarding. *Cisco Application Note*, 1998.

[7] J. T. Dixon and K. L. Calvert. Increasing Demultiplexing Efficiency in TCP/IP Network Servers. In *Proceedings of the Internation Conference on Computer Communication Networks*, October 1996.

[8] J. Duncanson. Inverse Multiplexing. *IEEE Communications Magazine*, April 1994.

[9] P. Fredette. The Past, Present and Future of Inverse Multiplexing. *IEEE Communications Magazine*, April 1994.

[10] International Organization for Standardization. Information Processing Systems — Data Communication High-Level Data Link Control Procedure — Frame Structure. *ISO 3309*, October 1984.

[11] R. Jain. A Comparison of Hashing Schemes for Address Lookup in Computer Networks. *IEEE Transactions on Communications*, October 1992.

[12] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, MA, 1973.

[13] J. Moy. OSPF Version 2. *RFC 2328*, April 1998.

[14] D. Thaler. Multipath Issues in Unicast and Multicast. *Internet Draft, http://www.ietf.org/internet-drafts/draft-thaler-multipath-01.txt*, November 1998.

[15] D. Thaler and C. Ravishankar. Using Name-Based Mappings to Increase Hit Rates. *IEEE/ACM Transactions on Networking*, February 1998.

[16] C. Villamizar. Comparison of OSPF-OMP and Direct Connection. *http://engr.ans.net/ospf-omp/ramp.html*, October 1998.

[17] C. Villamizar. OSPF Optimized Multipath (OSPF-OMP). *Internet Draft, http://www.ietf.org/internet-drafts/draft-ietf-ospf-omp-01.txt*, October 1998.