Cost-Effective Flow Table Designs for High-Speed Routers: Architecture and Performance Evaluation

Jun Xu (contact author)
College of Computing
Georgia Institute of Technology
jx@cc.gatech.edu

Mukesh Singhal
Department of Computer Science
University of Kentucky
singhal@cs.uky.edu

February 28, 2002

Abstract: Provision of QoS-related router functions such as traffic regulation, policy routing, and usage-based accounting requires that a flow table stores state information for active flows. The design of such a flow table is not trivial for a high-speed Internet router (e.g., 100+ Gbps) with a large number of active flows (e.g., tens of millions) and a high packet arrival rate (e.g., tens of millions of packets per second). Targeting two different models (centralized and distributed) of router design, we propose a software-based design to be implemented on individual line cards, which is suitable for the distributed model, and a hardware-based design to be implemented in the main forwarding engine of a router, which is suitable for the centralized model. The software-based design, adapted from hash table data structure, employs a practical and effective technique to solve the garbage collection problem caused by the expired flows. The hardware-based design, adapted from the architecture of an N-way set-associative cache, employs a dynamic set-associative scheme to reduce the overflow ratio that traditional set-associative scheme incurs, by a high percentage, and a pipelined design to achieve a throughput of 100+ Gbps. The performance evaluation results from both trace-driven simulation and statistical analysis demonstrate that both designs are cost-effective for their targeted router models.

Index Terms: Flow Table, Performance Analysis, Router Architecture, Universal Hashing.

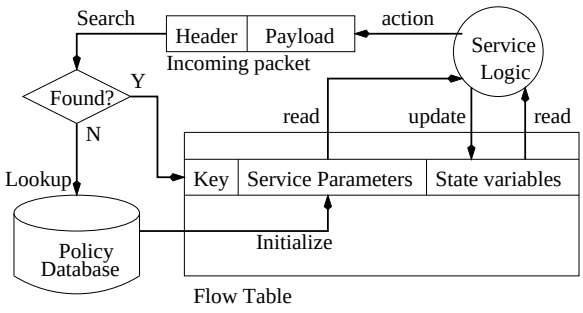

Figure 1: Operation Flowchart of a Flow Table

1 Introduction

Maintaining a flow table in an Internet router is essential for the provision of QoS-related router functions such as *usage-based service accounting* [1], *traffic regulation* [2], and *policy routing* [3]. For traffic regulation, for each *active flow*, the flow table keeps a *timestamp* that indicates whether the flow is conforming to its *negotiated profile*. For policy routing, the flow table caches the next-hop and QoS information to be applied to the packet, which is obtained from a relatively expensive *policy lookup* process [3, 4, 5], so that one lookup is needed per *flow*, not per packet. For usage-based accounting, the flow table uses a flow instead of a packet as the basic granule for accounting, which typically reduces the amount of work by about 90 percent [6]. The overview of a flow table is shown in Fig. 1. The schema of a flow table consists of *service parameters* and *state variables* used in service accounting (e.g., unit price), traffic regulation (e.g., timestamp), and policy routing (e.g., next-hop). The header fields of an incoming packet will be used as the key to search for the flow that the packet belongs to. If such a flow entry is found, the service parameters and state variables in the entry will be fed to the service logic, which determines the *action* to be performed (e.g., which output port to route the packet) on the packet and the new values the state variables

(e.g., timestamp) should be updated with. We refer to such a "search-compute-update" cycle as a *flow transaction*. If a matching entry is not found, it indicates that this packet is the first packet of a new flow. Then an entry needs to be created in the table for the flow, and a *lookup* to the *policy database* is needed to determine the service parameters the entry should be initialized with.

The flow table is a part of the forwarding engine in a QoS router. So its design is different for two router models that make drastically different design choices on the functional requirement and placement of the forwarding engine, namely, *centralized router model* and *distributed router model* [7]. In the centralized model, when a packet arrives at a line card, its header will be forwarded to a central forwarding engine through the *switching fabric*. The forwarding engine will decide the next-hop to route the packet and its QoS if applicable, and send the decision back to the line card. In the distributed model, however, each line card has its own forwarding engine to make such decisions. The advantages and disadvantages of both models and the commercial router products that adopt each model are discussed in detail in [7]. Two router models place drastically different throughput requirements on the flow table. In the centralized model, one flow table has to handle all the flow transactions of the router, while in the distributed model, a flow table only needs to handle the flow transactions at a line card.

Targeting these two router models, we proposed two cost-effective flow table designs: a software-based design (5 Gbps throughput) for the distributed model, and a hardware-based design (100+ Gbps throughput) for the centralized model. The major challenge in the software-based design, which employs hash table data structure, comes from the need to purge *expired flows* from the flow table. We find that *garbage collection* with hash table data

structure in real time is not a trivial task. The software-based design employs our amortized garbage collection technique that "absorbs" the overhead of garbage collection into that of probing and achieves a nice tradeoff between memory utilization and throughput. The major challenge in the hardware-based design is to achieve high throughput at relatively low cost. Adapted from the architecture of N-way set-associative cache, the hardware-based design employs our *dynamic set-associative* scheme that reduces the overflow inherent with set-associativity by a high percentage. Employing relatively inexpensive yet high-speed DRAM chips (e.g., 800 MHz RAMBUS [8]), it nicely pipelines the system to "absorb" the DRAM access latency and other system overhead so that a flow transaction can be completed very quickly (e.g., every 20 ns).

In the following, Section 2 introduces concepts related to a flow and states important assumptions. Section 3 and 4 present the architecture and performance evaluation of the software-based design and the hardware-based design respectively. Related work is surveyed in Section 5. Section 6 summarizes the main contributions of the work. As the focus of this paper is on how to design a flow table to support the aforementioned router functions, the detail of these functions and their implementation are omitted in this paper in the interest of space. They can be found in the longer version [9] of this paper.

2 Flow Concepts and Important Assumptions

A *flow*, as shown in Fig. 2, is defined [10] as a series of unidirectional packets that share the same <src_IP, dest_IP, src_port, dest_port, protocol> tuple, separated by no more than a tunable timeout value D_{expire}. Here fixed timeout value (D_{expire}) is adopted to simplify

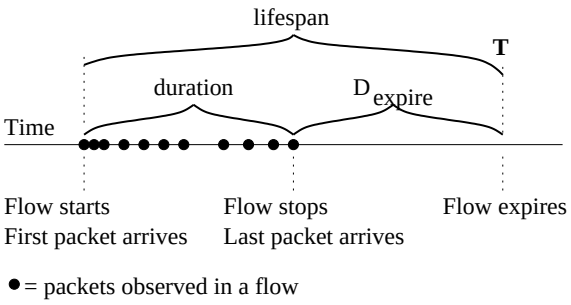

Figure 2: Timeline of a Flow

implementation. We will subsequently refer to such a tuple as *layer-4 address*. A flow *starts* when its first packet arrives and *expires* when the flow has no activity for a timeout period D_{expire}. The flow is *presumed active* in the meantime and this period is called the *lifespan* of the flow. Suppose the *expiration* happens at time T, the flow actually *stops* at time $T - D_{expire}$, when the last packet of the flow arrives. The period between the time the flow starts and the time it stops is called the *duration* of the flow, during which the flow is *actually active*.

Aggregate network traffic is found to be bursty and self-similar [11, 12], and possible reasons behind that are analyzed in [13]. However, to the best of our knowledge, the arrival of new flows and the number of active flows (when D_{expire} is reasonably long, say 60s, to prevent a flow from being segmented into multiple subflows), at the Internet backbone level, have never been found to be self-similar. Actually, we don't expect them to be self-similar: while packets inside a connection are correlated, arrivals of the new connections (flows) are generally not [14].

Important metrics in the software- and hardware-based designs are modeled under a *context* called *dynamically-balanced flow system* (DBFS). In DBFS, we assume that, every

$\frac{1}{NF}$ seconds, one new flow arrives and one old flow expires. In other words, during every one-second interval, exactly NF old flows expire and NF new flows arrive so that the number of *active flows* AF remains constant over time. DBFS assumption is close to the reality as it is shown in [10, 15] that AF tends to fluctuate very little in a 5-minute interval, though it may change dramatically over a one-day period. As we have explained, DBFS does not contradict with the self-similar nature of the Internet traffic [11, 12]. We understand that DBFS modeling does not capture the "disturbance" caused by the fluctuation of the active flows in the real traffic. However, the agreement between the analytical (from the modeling) and the experiment (from the simulation) results suggests that the effect of this "disturbance" on the metrics we are interested in is very small.

Poisson modeling that will occur in several places of this paper also needs to be carefully justified. In all but one place, the Poisson arrivals and departures are assumed in a slot of a hash table or in a line of a set-associative cache (similar to the former case). The justification is that under the DBFS context and the uniform hashing assumption, the interarrival time between two new flows that are mapped to a particular slot/line is accurately modeled as a geometric random variable. So, when the number of lines/slots is large, this geometric distribution can be approximated as the Poisson distribution. Only in one place (in Section 3.1) do we assume that the interarrival time between packets inside a flow is exponential (with parameter I). However, we immediately show that our targeted metric is very insensitive to the variation of I.

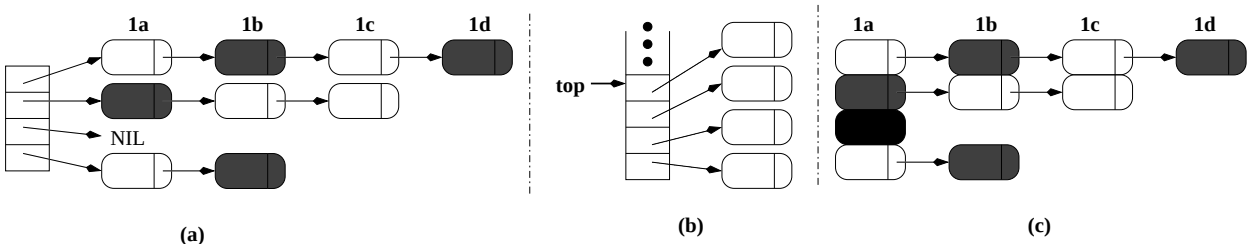

Figure 3: (a) Chaining with head table, (b) Stack of free nodes, (c) Chaining without head table.

3 Software-based Design

In this section, we present a software-based design that achieves a throughput of 5 Gbps, which is suitable for a line card. This design is based on hash table data structure augmented with a technique to solve the *garbage collection* problem. When a flow expires, the hash node that contains the flow entry becomes "garbage." Without a "garbage collection" mechanism, the system will eventually run out of memory. We develop an amortized garbage collection technique that absorbs the overhead of purging into *probing*. We introduce a metric called *garbage ratio* that measures the effectiveness of the scheme. Modeling and simulation results on this metric agree to within 10% difference.

3.1 Amortized Garbage Collection (AGC)

The data structure underlying AGC is *hashing with chaining*[1] as shown in Fig. 3(a). The nodes in the hash table are 32-byte long flow entries. A hardware hash function that achieves *uniform hashing* is used to produce an index and its orthogonal part called *tag*, for each incoming layer-4 address. Such a suitable hash function is described in Section 4.2. The

[1]We do not use *open hashing* algorithms because they make garbage collection a complicated task [16].

index is used to locate *a slot* in the *head table* that points to a linked list of nodes. A *probe* is performed along the linked list to see if the same tag is found in any of these nodes. The white nodes stand for active flows and the gray nodes stand for expired flows. In the following discussion, we use words "node" and "flow" interchangeably.

The AGC scheme works as follows. While the probe is carried out along the linked list, if the current node is found expired, it is deleted from the linked list and returned to a stack of *free nodes* (shown in Fig. 3(b)). For example, in Fig. 3(a), suppose a packet that belongs to flow 1c arrives. During the probing, AGC will find that node 1b has expired and will delete it from the chain. Node 1d (also expired), however, will not be affected because it is behind the node 1c. In other words, the purging occurs only when it is "convenient."

3.2 Performance Analysis of AGC

The effectiveness of the scheme is measured by the *garbage ratio GR*, the number of *expired flows* that remain in the hash table divided by the total number of active flows AF. This is precisely the "gray/white" ratio in Fig. 3(a). Under the DBFS context, the garbage ratio GR is modeled as follows. Let $T_{lifespan}$ be the average lifespan of a flow, which is also the average lifespan of a white node. Then according to *Little's law* [17], $AF = T_{lifespan} * NF$. Let $\hat{T}_{purge}$ be the latency between the time a node expires and the time it is purged by AGC, and let T_{purge} be its mean. So T_{purge} is the average lifespan of a gray node. Again according to Little's law, the number of "gray nodes" is $T_{purge} * NF$. So the garbage ratio ("gray/white") is

$$GR = \frac{T_{purge} * NF}{T_{lifespan} * NF} = \frac{T_{purge}}{T_{lifespan}} \tag{1}$$

Under DBFS context, $T_{lifespan}$ is constant over time. So, to calculate GR, it remains to model T_{purge}. Note $T_{lifespan} - D_{expire}$ is the average duration of a flow, during which the flow is *actually active*. Again by Little's law, the ratio of the *presumed active flows* that are *actually active* is $\frac{T_{lifespan}-D_{expire}}{T_{lifespan}}$. We call this ratio the *actually active ratio*, and denote it as ρ. In the following discussion, unless the word "actual" is explicitly used, the term "active flows" refers to "presumed active flows." Suppose the number of *slots* in the hash table is M. Define the *load factor* LF as AF/M. Let us consider an arbitrary slot. As explained in Section 2, when M is large, the arrival process can be approximated as Poisson so that $P[\hat{U} = i]$=$exp(-LF) * LF^i/(i!)$. So among M slots of the hash table, approximately $M * P[\hat{U} = i]$ of them contain i active nodes, i=0,1,2,..., and among AF active flows, approximately $i * M * P[\hat{U} = i]$ nodes belongs to a slot that contains i active nodes, i=0,1,2,... Then randomly pick a flow among AF active flows, the number of active nodes (denoted as $\hat{V}$) in the slot that the picked node belongs to is of the following distribution: $P[\hat{V} = i] = \frac{i*M*P[\hat{U}=i]}{AF} = \frac{i*exp(-LF)*LF^i}{LF*(i!)}$, where $i = 1, 2, ...$ Though this distribution is derived using informal analysis, it can be shown that this is precisely the *limiting distribution* when both M and AF go to infinity and $AF/M = LF$. Suppose the picked node is the $\hat{W}_{th}$ active node along the linked list. Since each node among those $\hat{V}$ nodes is equally likely to be picked, the joint distribution of $\hat{V}$ and $\hat{W}$ is

$$P[\hat{V} = i, \hat{W} = k] = \begin{cases} \frac{exp(-LF)*LF^i}{(i!)*LF} & : \quad k \leq i \\ 0 & : \quad k > i \end{cases}$$
$$i, k = 1, 2, \cdots \tag{2}$$

Now we are ready to analyze T_{purge}. Suppose exactly at time 0, a node ND expires, which can be any of the AF nodes with equal probability $1/AF$. Suppose the slot that ND belongs to has i nodes and ND is the k_{th} along the probing sequence ((2) tells us the probability of such an event). Suppose among the $i-k$ active nodes behind ND, $\hat{X}$ nodes are *actually active*. $\hat{X}$ is of binomial distribution $P[\hat{X}=j]=\binom{i-k}{j} * \rho^j * (1-\rho)^{i-k-j}$,$j$=1,2,...,$i-k$. ρ is the aforementioned *actually active ratio*. There are two possible ways in which the expired node gets evicted. The first one occurs when one of those $\hat{X}$ actually active nodes get a *visitor* (an incoming packet that belongs to the flow). The second occurs when a new flow arrives and is hashed to ND's slot. For each of the j actually active nodes, the arrival time of the first *visitor* to the node (denoted as $\hat{T}_1$, $\hat{T}_2$, ..., $\hat{T}_j$) after time 0 can be modeled as an exponential distribution with mean I. I is the average interarrival time of packets in a flow, modeled as $(T_{lifespan} - D_{expire})/NP$. NP is the average number of packets in a flow. Let $\hat{T}_{new}$ be the arrival time of the first new flow that is hashed to ND's slot after time 0 and T_{new} be its mean. It can be shown that $\hat{T}_{new}$ is of exponential distribution with mean $T_{lifespan}/LF$ when M is large. Then

$$\hat{T}_{purge} = MIN(\hat{T}_1, \hat{T}_2, ..., \hat{T}_j, \hat{T}_{new}) \tag{3}$$

Given n independent exponential random variables $\hat{Y}_1$,$\hat{Y}_2$,...,$\hat{Y}_n$ with mean λ_1,λ_2,...,λ_n respectively, MIN($\hat{Y}_1$, $\hat{Y}_2$, ..., $\hat{Y}_n$) is also of exponential distribution and its mean is $1/(\sum_{m=1}^{n} \frac{1}{\lambda_m})$. So $\hat{T}_{purge}$ (conditioned on j) is also of exponential distribution, and its mean is $1/(\frac{j}{I} + \frac{1}{T_{lifespan}})$. Piecing all these formulae together, we obtain our main result:

$$E[\hat{T}_{purge}] = \sum_{i=0}^{\infty} \frac{i * exp(-LF) * LF^i}{LF * (i!)} * \sum_{k=1}^{i} \frac{1}{i} * \sum_{j=1}^{i-k} \binom{i-k}{j} * \rho^j * (1-\rho)^{i-k-j} * \frac{1}{\frac{j}{I} + \frac{LF}{T_{lifespan}}} \quad (4)$$

Once we have calculated $E[\hat{T}_{purge}]$, the garbage ratio GR can be obtained as $\frac{T_{purge}}{T_{lifespan}}$ according to (1). Parameters contained in (4) include LF, ρ, I, and $T_{lifespan}$. Since $T_{lifespan}$ can be derived from ρ and D_{expire}, given a fixed D_{expire}, GR is a function of three independent parameters I, ρ, and LF. An analysis of (4) shows that the garbage ratio GR increases when LF decreases and vice versa. Plugging in the parameters (I=0.95s, ρ=0.25) measured from the FIXWEST trace 1 when D_{expire} is set to 60s, the GR calculated using (4) and (1) is 0.89, 0.40, and 0.16, when LF is 1, 2, and 4, respectively. So to reduce this GR, we would like to increase LF. However, larger LF leads to longer *probe sequence* because the average probe length is $1 + LF/2$ when the node is found in the table and is $1 + LF$ when the node is not found [16].

I and ρ are traffic-dependent parameters, but fortunately GR is not sensitive to changes of these two parameters. Otherwise, GR would be traffic-dependent. In all three aforementioned cases (LF=1,2,4), when ρ is doubled/halved, GR is decreased/increased by at most 0.04, and when I is doubled/halved, GR is increased/decreased by less than 0.005. Actually, there is a simple yet rough bound of the GR that is determined only by LF. From (3), we know that $\hat{T}_{purge} \leq \hat{T}_{new}$. So $T_{purge} \leq T_{new} = T_{lifespan} * LF$. So the garbage ratio $GR = T_{purge}/T_{lifespan} \leq 1/LF$. This upper bound is 1, 0.5, and 0.25, when LF is 1, 2, and 4, respectively. This is not very far away from the results (0.89, 0.40 and 0.16) obtained using (4).

Trace	1	2	3	4	5	6	7
Date	6/21/95	2/28/96	9/18/96	9/26/96	1/9/97	5/17/97	11/20/97
Duration(sec)	1625	770	298	1233	1084	298	1155
Avg. Pkts/s	14414	13625	13939	9995	8088	3856	9494

Table 1: Statistics of the FIXWEST Traces [18] Used in the Simulation

3.3 Simulation Study of AGC

We conducted a simulation study to independently validate (4). Seven publicly available FIXWEST backbone traces [18] (see Table 1) were used in the simulation. The results are presented in Fig. 4. Fig. 4(a), 4(b), and 4(c) show AF in FIXWEST trace 1, 2, and 4, respectively. The number of slots M in all three cases is 16384. D_{expire} is set to 60s, 30s, and 40s, respectively, to adjust AF so that $LF = AF/M$ varies between 3 and 4. Fig. 4(d), 4(e), and 4(f) compare the Analytical GR with the experimental GR in these three traces. Each point on the analytical GR curve is calculated using (4) according to $LF = AF/M$ at that second and the traffic parameters (I and ρ) of the corresponding trace. The experimental GR is measured on a per second granule using a simulation program. So this is a "point-wise stress test." In all three cases, the difference between these two curves is no more than 10 percent. This shows that the analytical modeling estimates GR very well.

Implicitly, the existence of the *head table* (see Fig. 3(a)) makes our discussion much easier. However, it does not make sense in the real implementation. It increases the probe length by one and wastes about 10% of the memory (each pointer is 3 byte long). Now it is time to remove it. Fig. 3(c) shows the aftermath of the removal of the head table from the

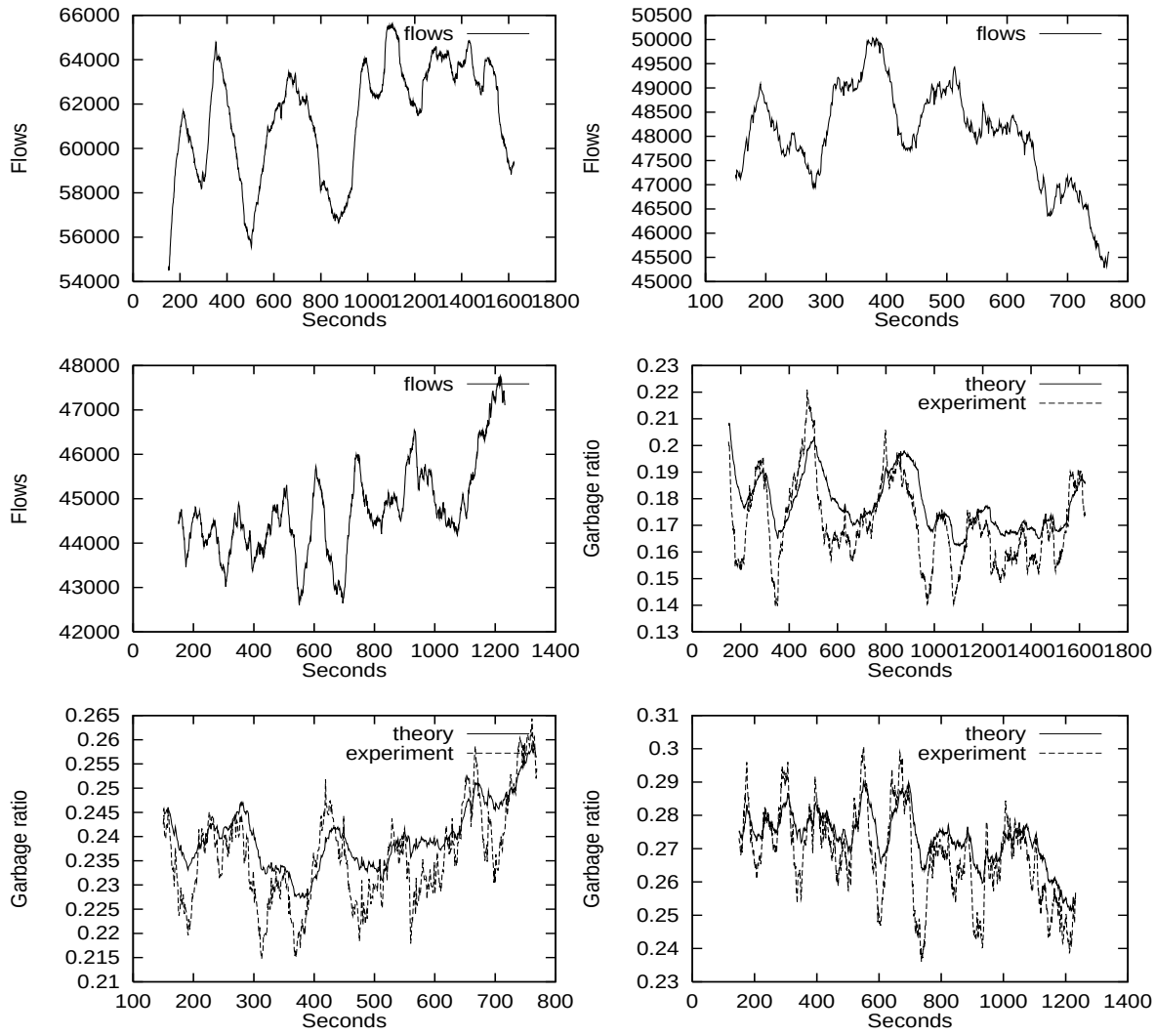

Figure 4: Analytical and Experimental Results (First Row: (a)(b), Second Row: (c)(d), Third Row: (e)(f))

hash table shown in Fig. 3(a). The black node in Fig. 3(c) is a node that does not contain a flow entry, but has to be there to indicate that no flow exists in this slot. Since such nodes do not exist in the hash table shown in Fig. 3(a), GR calculated using (4) does not count it in. It can be shown that the "black/white" ratio is $exp(-(GR+1)*LF)/LF$. When LF is 4, this contributes less than 0.02 to the total garbage ratio. Also, the *free node stack* (Fig. 3(b)) adds an additional 10 percent overhead (to store pointers). So the total memory overhead is 0.16+0.02+0.10=0.28, that is, the total data memory (excluding program code) the system needs is 1.28 times of the memory needed to store the AF active flows.

3.4 The Throughput of the Software-based Design

We estimated that the software-based design can achieve a throughput of at least 5 Gbps (a little higher than OC-96) using the combination of a low-end general-purpose CPU and a high-end DRAM (RAMBUS) chip. To achieve 5 Gbps throughput, the average execution time of a flow transaction must be no more than 400 ns, assuming an average packet size of 250 bytes (a widely-used assumption on the packet size). Suppose that commercially available 16-bit-wide 800 MHz RAMBUS are used. This allows 16 bytes (the minimum *burst size*) to be read or written every 10 ns in *burst mode* (a series of accesses to consecutive memory locations), after an initial *access latency* (typically 60 ns). So it takes a total of 80 ns to access a 32-byte flow entry. If we assume that LF is 4, the probe length (if found) will be 3 (=LF/2+1) as explained before. Each probe will take about 100 ns (including instructions to "prepare to read"). We can not overlap these probe accesses because the memory address (the pointer) needed to access the next node in the chain is not known until

the current node is read. This is the reason why we have to use high throughput DRAM such as 800 MHz RAMBUS. However, during such an 100 ns memory access period, the CPU does not sit idle. AGC overlaps the execution of the instructions that check whether current node expires or matches the packet header with the memory access of the next node. In this sense, the "purging" overhead is absorbed into the "probing" operation. We coded AGC using DLX [19] assembler and optimized it to minimize the number of stalls that will occur in the pipelined execution. We estimated that 200 MHz clock rate is good enough for us to achieve 400 ns average execution time.

4 The Hardware-based Design

Though the software-based design is fast enough for a line card, it is too slow for the forwarding engine in a high-end centralized router. In this section, we present a hardware-based design that can deliver a high throughput of 100+ Gbps.

As shown in Fig. 5, the main component of the hardware-based design is N independent memory banks, each of which stores up to $M = 2^r$ flow entries. Each bank is associated with a *different* hardware hash function, which hashes the 97-bit long layer-4 address of an incoming packet and produces two values. One is an r-bit hash value used as an *index* into the corresponding bank in order to locate a *candidate entry*, where the flow that the packet belongs to may potentially be stored. The other is a $97 - r$ bit long *tag*, which is the *orthogonal part* of the index[2]. Each tag is compared with the tag field contained in

[2]In computer architecture, a set-associative CPU cache simply extracts some of the memory address bits as the index (this hash function is called *bit extraction*) and the remaining bits (the orthogonal part) as tag.

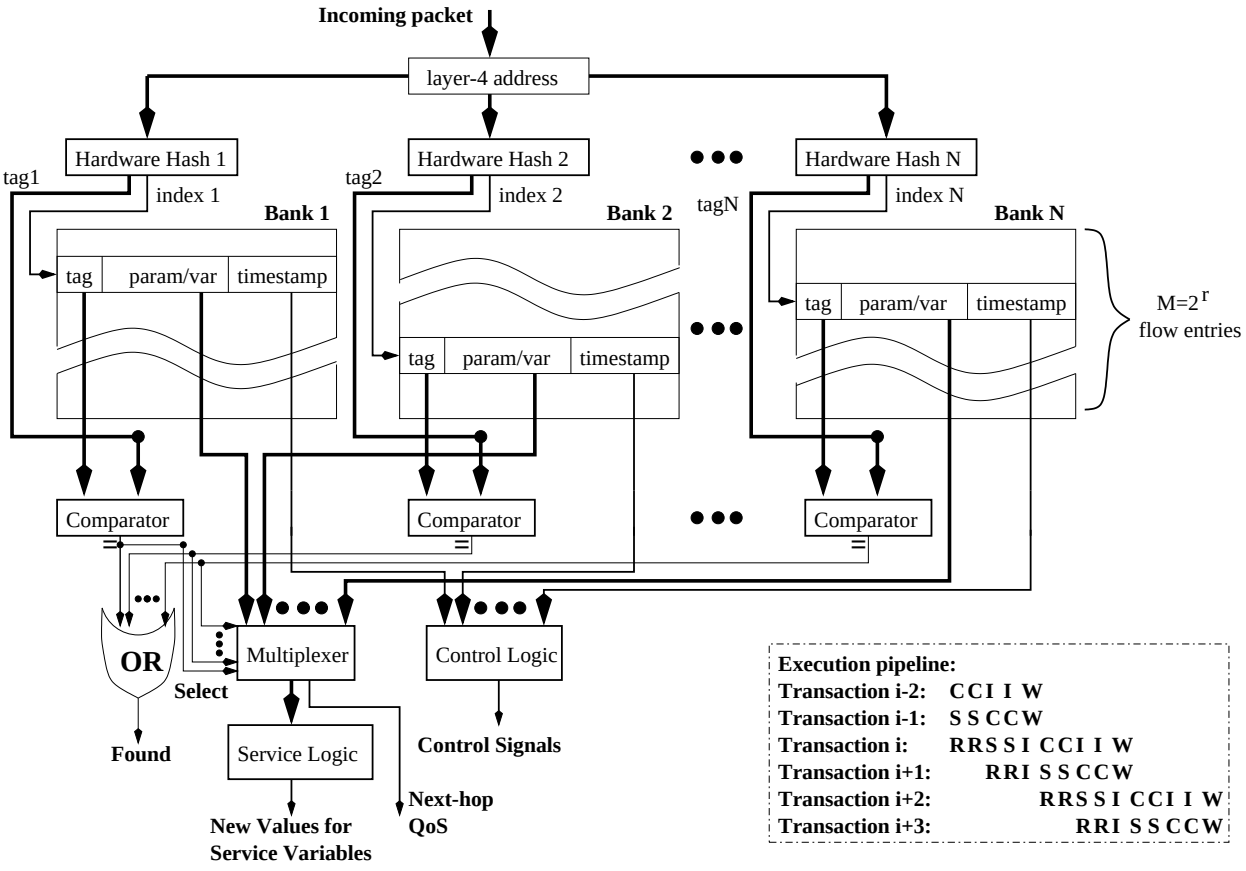

Figure 5: The architecture of the hardware-based design

the corresponding flow entry in parallel. If there is a match, we have a *hit* and the state variables and service parameters of the matched entry will be fed into the *service logic* from the *multiplexer*. Otherwise, we have a *miss*, and a new entry needs to be brought into the flow table. If one or more of these N candidate entries (called "incumbents") have expired (by checking the corresponding timestamp field), one of expired entries will be randomly chosen for replacement. This random replacement can be emulated in hardware using a small amount of ROM, as shown in [6]. In case no incumbent has expired, an *overflow* happens. Note that the operation of a flow table is different from that of a CPU cache: here we can not randomly kick out an incumbent that has not yet expired to give room for the incoming flow. So in the hardware-based design, the overflowed flow entries are "absorbed" into a software-based *assistant flow table*, the design of which we have just presented. However, since the software-based design is about an order of magnitude slower than the hardware-based design (called *master flow table*), it is desirable that the number of such overflow be as small as possible. To reduce the overflow, we propose a dynamic set-associative scheme (referred to as dynamic scheme thereafter) in which these N hardware hash functions, denoted as h_1,h_2,..., h_N, not only are different, but also satisfy the following *N-universal property*: given a random hash key X, $h_1(X)$,$h_2(X)$,$\cdots$,$h_N(X)$ are independent uniformly distributed random variables. In contrast, in a set-associative CPU cache (referred to as static scheme thereafter), the same hash function (typically a bit extraction) is used with each and every bank. Compared to the static scheme, dynamic scheme cuts the amount of overflow by about 50 to 75 percent, to be shown in Section 4.4.

4.1 Throughput Analysis

The system is pipelined to achieve a very high throughput (e.g., 100+ Gbps). The pipelining schedule is shown in Fig. 5. Each letter (R,S,C,W,I) denotes a clock cycle (10 ns or the amount of time to transfer 16 bytes using 16-bit-wide 800 MHz RAMBUS technology). R, W, and I stand for read, write, and idle, respectively. As we explained in Section 3.4, two clock cycles are needed to read an entry, and one clock cycle is needed to update an entry because more than half of the entries are read-only. S is the search ("comparator," "multiplexer," and "OR" in Fig. 5) and replacement ("control logic") logic. S logic is quite simple and can be completed well within two clock cycles (e.g., 20 ns) using 0.25 μm CMOS VLSI. C stands for the computation in the service logic. We explained in [9] that the service logic can also be completed within two clock cycles. Since in each transaction, W only occurs to one memory bank, *write merge* among consecutive transactions is used to improve the system throughput. As shown in Fig. 5, W in transaction i and W in transaction $i+1$ are executed simultaneously[3] so that two flow transactions are completed every five cycles. So each transaction incurs a base cost of 2.5 clock cycles. However, there is $1/N$ probability that these two writes occur to the same bank. In this case, a *stall* of one cycle will occur, resulting an amortized additional overhead of $\frac{1}{2N}$ clock cycle per transaction. Also the system needs to absorb the overhead of writing an "interim record" to address the read/write race condition when a miss occurs. The amortized overhead of this is $(2.5+\frac{1}{2N})r$ clock cycles per transaction, where r is the miss ratio. Therefore the gross total processing per transaction is $(2.5+\frac{1}{2N})*(1+r)$ clock cycles. When N=6 (the recommended minimum size) and the miss

[3]Merging three or more writes into a cycle can further improve the throughput, at the cost of higher hardware complexity.

ratio is 10%, the throughput of the architecture is estimated to be one transaction per 2.85 (=(2.5+1/12)*(1+0.1)) clock cycles. When each clock cycle is 10 ns, this is equivalent to 35 million transactions per second or 70 Gbps when the average packet size is 250 bytes. Note that this bandwidth can be improved to 140 Gbps if future generation 1600 MHz RAMBUS is used (the size of a flow entry or 32 bytes is exactly the minimum burst size of the 1600 MHz RAMBUS).

4.2 The Hash Function

The hash functions for the dynamic set-associative scheme have to satisfy, in additional to the aforementioned N-universal property, a number of other nice properties: (a) It should be able to generate a hash key within a clock cycle[4]; (b) The orthogonal part of the hash key should be well-defined and should also be computed within 10ns. Such hash functions are hard to find: (a) Bit extraction won't work because the bits extracted from a layer-4 address are generally not random, (b) Traditional linear congruent hash functions such as $(a * x + b) mod(c)$ are too slow and its orthogonal part in general is not defined; (c) CRC or Rabin (and its variants) hash functions, which involves multiplication, addition and modular operations on polynomials with binary coefficients, are too slow for this 10ns mission because the critical timing path of each hash operation would consist of at least 97 serial cyclic shifts and XOR operations (a layer-4 address is 97-bits long).

We found that a class of hash functions called H_3 [20] has the aforementioned desired properties. It was shown in the literature that N hash functions randomly chosen from H_3 satisfy the N-universal property. Though H_3 has been discovered for more than 20 years

[4]When 1600 MHz RAMBUS is used, a hashing operation has to be completed within 10 ns.

[20], its unique advantage in terms of both speed and well-definedness of the orthogonal part over other hash functions is recognized by us. An H_3 hash function maps a w-bit binary string $A = a_1 a_2 \cdots a_w$ to an r-bit binary string $B = b_1 b_2 \cdots b_r$ by the following linear transformation:

$$\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_r \end{pmatrix} = \begin{pmatrix} q_{11} & q_{12} & \cdots & q_{1w} \\ q_{21} & q_{22} & \cdots & q_{2w} \\ \cdots & \cdots & \cdots & \cdots \\ q_{r1} & q_{r2} & \cdots & q_{rw} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_w \end{pmatrix}$$

Here, Q is a $r \times w$ matrix defined over GF(2)=$\{0,1\}$. Each hash function in H_3 corresponds uniquely to such a Q. The multiplication and addition in GF(2) are boolean AND (denoted as $\circ$) and XOR (denoted as $\oplus$), respectively. So, each bit of B is calculated as: $b_i = (a_1 \circ q_{i1}) \oplus (a_2 \circ q_{i2}) \oplus \cdots \oplus (a_w \circ q_{iw}) \quad i = 1, 2, \cdots, r.$

The orthogonal part of B^T is obtained as follows. Denote the row vectors of Q as V_i, $i = 1, 2, \cdots, r$. We restrict the choice of V_i so that they are *linearly independent*. This is not a severe restriction because it can be shown that, when w=97 and $r < 24$, the probability for r vectors randomly chosen from $\{0,1\}^w$ being *linearly dependent* is very small (< 0.000001). Then we can expand these r vectors into a basis V_1, V_2,$\cdots$,V_r,V_{r+1},V_{r+2},$\cdots$,V_w in the vector space $\{0,1\}^w$. Let $\overline{Q}$ be the $(w-r) * w$ matrix, the i_{th} row of which is V_{r+i}, i=1,2,...,$w-r$. Then $\overline{B}^T = \overline{Q}A^T$ is the orthogonal part of B^T. Implemented in hardware, the hashing operation can be completed well within 10ns because the critical timing path for generating each and every bit is of the same length, which consists of log(N) (here N=97) levels of XOR operations.

4.3 Implementation and Cost Analysis

The H_3 hash function is very amenable to hardware implementation. We estimated that a 97-in 97-out H_3 hash function occupies an area of 2.5mm by 2.5mm using 0.25 μm CMOS process. Such a chip will be low in cost (e.g., 5 dollars), assuming reasonable level of mass production (e.g., 3000+). This again justifies the dynamic set-associative scheme: the only difference in implementation between the dynamic scheme and the static scheme is these hash functions. This is well compensated by the amount of memory saved from the assistant flow table due to much smaller overflow ratio.

The major cost of the flow table comes from memory, the amount of which is proportional to the total number of active flows. Assume that the number of active flows grows at most linearly with the traffic volume[5]. It can be extrapolated, based on the traffic statistics measured at MCI vBNS backbone nodes (http://www.vbns.com/), that the total number of active flows for a 100 Gbps router is no more than 16 million (with D_{expire} set to 60s). This translates into 512 MB of table entries, and 730 MB of memory requirement, with a "comfortable load" of the system at 70%. Though there is some indication and intuitive arguments [6] that this linear growth model is reasonable, we have so far not been able to empirically validate this model since traffic traces at the rate over 10 Gbps is not available. Nevertheless, since the price of memory per MB drops quickly, the cost of memory will still be very reasonable (e.g., within 1000 dollars) even if the actual memory requirement (for a 100+ Gbps router) deviates considerably from our estimate (e.g., doubling our estimate). Such a cost is well justified for a 100+ Gbps QoS router.

[5] Analytical arguments that justify this assumption can be found in [6].

4.4 Performance Evaluation: Dynamic vs. Static

In this section, we show that the overflow ratio under the dynamic scheme is much smaller than that under the static scheme. A flow that is "absorbed" into the assistant flow table may have a chance to *relocate* into the master flow table if one among the N candidate entries that the flow was "bidding for in futile" now expires. This relocation is "packet-driven" in the sense that it may occur only when an incoming packet that belongs to the flow triggers a miss in the main flow table. Whether or not the relocation is allowed can be viewed as a system design option ("with relocation" vs. "without relocation"). In reality, relocation is always preferred because it reduces the amount of overflow in both the dynamic scheme and the static scheme, with little extra design complexity. However, statistical analysis of the overflow ratio becomes hard. On the other hand, the overflow of both static and dynamic schemes "without relocation" can be accurately modeled, the result of which is important in two aspects. First, the overflow from dynamic/static scheme "without relocation" serves as the absolute upper bound of the overflow from dynamic/static scheme "with relocation." This adds a statistical guarantee to the empirical results. Second, since empirical results show that dynamic scheme and static scheme benefit almost "equally" from the relocation, the dynamic scheme remains better than the static scheme when relocation is allowed. In the following, we will statistically analyze the overflow ratios of the two schemes "without relocation" and verify them using trace-driven simulation. The empirical results on the overflow ratios of the two schemes "with relocation" will also be presented.

We define the *entry overflow ratio* (O_{entry}) as the number of flow entries in the assistant flow table divided by the total number of active flows AF. We define the *transaction overflow*

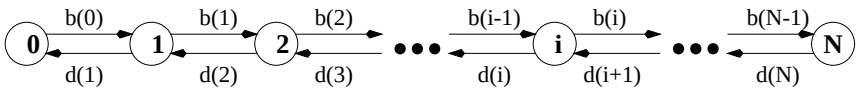

Figure 6: The Performance Modeling of the Static Set-Associative Scheme

ratio ($O_{transaction}$) as the number of flow transactions (defined in Section 1) that is resolved by the assistant flow table divided by the total number of flow transactions. When relocation is not allowed, we show that O_{entry}=$O_{transaction}$ by the following argument. Let $\hat{NP}$ denote the average number of packets in a flow. When the relocation is not allowed, $\hat{NP}$ averaged over the flows in the master flow table will be the same as the $\hat{NP}$ averaged over the flows in the assistant flow table, because the "bad luck" (a *Bernoulli random variable*) of a flow being overflowed to the assistant flow table is statistically independent of its $\hat{NP}$. As a consequence, the proportion of the active flows in the assistant flow table should be equal to the proportion of the transactions these active flows handle, that is, O_{entry}=$O_{transaction}$. So, in the following analysis, when the relocation is not allowed, we use O to denote both O_{entry} and $O_{transaction}$. Note that this equation does not hold when the relocation is allowed. Since relocation is packet-driven, flows in the assistant flow table with larger $\hat{NP}$ have a higher probability of relocation than flows with smaller $\hat{NP}$. So the flows that stay in the assistant flow table tend to have a smaller $\hat{NP}$, which results in $O_{entry} \geq O_{transaction}$.

We define *load ratio* of the master flow table LR as the number of active flows AF divided by its total capacity $M*N$. Under the aforementioned DBFS context, O (without relocation) in both dynamic scheme and static schemes is uniquely determined by LR and N as follows.

- Dynamic Set-Associative Scheme: Here we assume that a *dynamic balance* is achieved, in which the overflow ratio stays constant. We know that the load ratio of the master

flow table is $AF*(1-O)/(M*N) = LR*(1-O)$, which is also the *occupancy ratio* (by active flows) of each bank because under random replacement, these N banks should be equally loaded. Then, under the uniform hashing assumption, the probability for an incoming flow to be overflowed is $(LR*(1-O))^N$. Now consider a short time interval $[T, T+\Delta T]$. During this interval, $\Delta T * NF$ flows arrive and $(LR*(1-O))^N * \Delta T * NF$ overflows occur. These flow entries will be inserted into the assistant flow table. However, when the system is *dynamically balanced*, this number should be equal to the the number of expirations that occur in the assistant flow table, which is $\Delta T * NF * O$. When NF and ΔT are canceled out, the simplified equation is $(LR*(1-O))^N = O$. This equation has a unique solution in (0,1), which can be obtained through a numerical method.

- Static Set-Associative Scheme: We consider an arbitrary *line* (N entries with the same index) in the master flow table. Under the DBFS context and the uniform hashing assumption, when M is large, both the arrival (new flows) and the departure (expired flows) processes that occur in the line are approximately Poisson (explained in Section 2). So, the dynamics of the line can be modeled as a continuous Markov chain as shown in Fig. 6. The line is in state i if it contains i active flows. $b(i)$ and $d(i)$ are the arrival rate and expiration rate of the flows when the line is in state i. We know that $b(i) = NF/M$, which is the total flow arrival rate divided by the number of lines M. The expiration rate $d(i)$ is $i*NF/AF$ because every active flow has an equal opportunity to expire. Let $\hat{S}$ denote the number of active flows in this line. Then $N+1$ unknowns, namely, $P[\hat{S}=i]$, i=0,1,...,N can be solved from the following $N+1$ linear

	LR=0.6		LR=0.65		LR=0.7		LR=0.75	
	dynamic	static	dynamic	static	dynamic	static	dynamic	static
N=10	0.0057	0.043	0.012	0.060	0.022	0.079	0.038	0.10
N=9	0.0093	0.051	0.018	0.069	0.030	0.089	0.048	0.11
N=8	0.015	0.061	0.025	0.080	0.041	0.10	0.061	0.12
N=7	0.024	0.073	0.038	0.093	0.055	0.11	0.076	0.14
N=6	0.037	0.089	0.054	0.11	0.074	0.13	0.097	0.15

Table 2: Overflow ratio comparison between static and dynamic

equations: $\{P[\hat{S}=i]*b(i)=P[\hat{S}=i+1]*d(i+1) : i=0,1,...,N\text{-}1\}$ and $\sum_{i=0}^{N} P[\hat{S}=i]=1$. Then O is the probability that an incoming flow *sees* that the line is full. Since the arrival process is approximately Poisson, according to PASTA (Poisson Arrivals See Time Averages) [17], O is exactly $P[\hat{S}=N]$.

Table 2 compares O under static and dynamic schemes, when N varies between 6 and 10 and LR varies between 0.6 and 0.75. We observe that the overflow ratio of the dynamic scheme is typically only 1/4 to 1/2 of that of the static scheme. The difference is larger when LR becomes smaller and/or when N becomes larger.

4.5 Simulation Study

We conducted a simulation study using aforementioned FIXWEST traces (Table 1) to independently validate the analytical results of O. We obtained and investigated the overflow ratio (O_{entry} and $O_{transaction}$) when relocation is allowed. Fig. 7 shows the simulation results

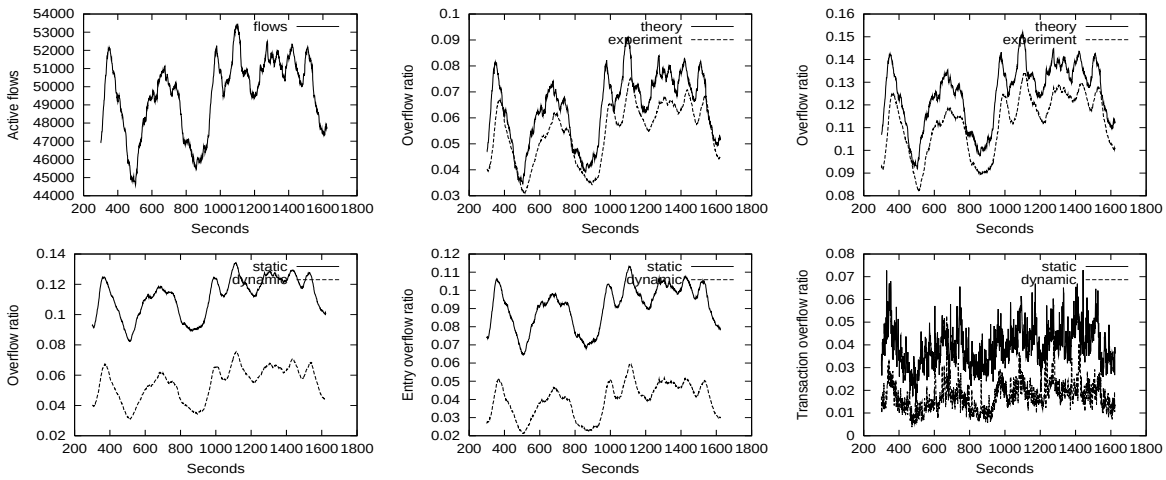

Figure 7: Analytical and Experimental Results (Top: (a)(b)(c), Bottom: (d)(e)(f))

obtained from trace 1 (Other traces produced similar results). Here, N is set to 8 and M is set to 8096. D_{expire} is set to 45 seconds so that the load ratio fluctuate between 60% and 80%. Fig. 7(a) shows the fluctuation of AF over the 1625s period. Fig. 7(b) compares the analytical O with the experimental O in the dynamic scheme (without relocation). Fig. 7(c) compares the analytical O with the experimental O in the static scheme (without relocation). In both Fig. 7(b) and Fig. 7(c), each point on the analytical curve is calculated using the aforementioned formulae, by plugging in the LR at that second. So, this is a "point-wise stress test." In both cases, the difference between the analytical result and the experimental result is consistently no more than 10%, which can be attributed to the fluctuation of AF that makes the reality deviate a little bit from the DBFS context. The fact that experimental result is consistently smaller is not surprising because it can be shown analytically that both the schemes exhibit the following "shock-absorbing" behavior: when the LR increases from a to b ($a < b$) and then decreases back to a, the overflow ratio during this period will be less than O, estimated under DBFS context and $LR = b$. Fig. 7(d) compares the experimental

O under dynamic scheme with the experimental O under static scheme (without relocation). The former is roughly 50% smaller than the latter, which agrees with the analytical results in Table 2. Fig. 7(e) and Fig. 7(f) show experimental results when the relocation is allowed. Fig. 7(e) compares O_{entry} under dynamic scheme with O_{entry} under static scheme. Fig. 7(f) compares $O_{transaction}$ under dynamic scheme (lower cloud) with $O_{transaction}$ under static scheme (higher cloud). Fig. 7(e) and 7(f) show that the dynamic scheme and the static scheme benefit almost equally from the relocation (about 20% for O_{entry} and 50% for $O_{transaction}$ in both dynamic scheme and static scheme). So in both Fig. 7(e) and 7(f), when the relocation is allowed, the overflow (O_{entry} or $O_{transaction}$) under the dynamic scheme is again roughly 50% of that of the overflow under the static scheme. This 50% reduction in $O_{transaction}$ makes a difference because the assistant flow table is typically 20 times slower (5 Gbps vs. 100+ Gbps).

5 Related Work

At the time of writing, Cisco's NetFlow was the only implementation of flow table on routers. NetFlow supports up to 128K flows [21] and typically operates at a rate less than 1 Gbps. As this implementation is proprietary in nature, neither its underlying data structure (or architecture) nor its garbage collection technique is published in open literature.

Garbage collection [22] has been an active research topic in programming languages for more than three decades. However, its context is memory management in functional language (e.g., LISP) and object-oriented language (e.g., Smalltalk) programming environment, where explicit allocation and deallocation of memory is undesirable. The main difficulty there is

to identify and locate the garbage, and maintain a data structure in order to eventually remove/collect them in a correct and efficient manner. In our scheme, on the other hand, the semantics in identifying and locating the garbage is relatively straightforward[6]. Our main contribution is to accurately analyze the performance and storage tradeoff of the garbage collection scheme based on the stochastic model we develop for the flow table system. To the best of our knowledge, no similar work exists in garbage collection literature [22] that focus on the accurate stochastic modeling of this tradeoff.

Work proposed for a different application in [23] bears some similarity to the dynamic set-associative scheme. A variant of Rabin hash function is used in that scheme because its speed requirement is much lower (at the microseconds level). Its targeted application and operation mode (and hence the performance modeling techniques) are totally different. Similar scheme has also been employed in our prior work [6], for a different application in which eviction of the incumbent entries that have not yet expired is allowed. This makes the targeted metrics and performance modeling techniques in both schemes quite different. Also, due to different system requirements, throughput optimization technique used in this paper is very different from the technique used in [6].

There are several other parallel hashing schemes in the literature that explicitly or implicitly employs N different N-universal hash functions [24, 25, 26, 27, 28] to reduce the metric their applications is targeting at. The application of [24, 25, 27] is dictionary or perfect hashing, which is not directly related to our scheme. Scheme introduced in [26] can be viewed as a type of Bloom filter [28], which pursues the universal hash functions in a quite different direction.

[6]Otherwise, a backgournd process, rather than garbage collection in the real-time, would be more efficient.

In another direction, skewed set-associative cache [29], independently proposed in the area of computer architecture, bears some resemblance to the proposed scheme. Skewed set-associative cache also employs different hash functions to generate indices in different cache banks. However, the hash functions used in skewed set-associative cache are very simple. They differ from bit extraction only in one bit, which is generated by XORing two or three address bits. In CPU caching, this simplicity is justified: this hash operation is on the critical timing path of the cache access cycle. However, the tradeoff of this simplicity is that it is far less aggressive in reducing collision miss as our dynamic set-associative scheme does due to the fact that hash indices have most of bits in common and therefore strongly dependent on each other. In comparison, our scheme allows the use of more sophisticated hash functions because the successive lookup requests are not dependent on each other. Also, no *analytical* performance evaluation has ever been performed on skewed set-associative cache and empirical evaluation is based on selected memory traces. In comparison, we have performed an accurate analytical performance evaluation of the miss ratio of the architecture, which is corroborated by a simulation study.

Content Addressable Memory (CAM), also known as fully-associative cache, has been used in bridges and switches for the fast lookup of the L2 address to output port mapping. CAM, however, is not suitable for implementing the flow table for two reasons. First, CAM is expensive (about $30 per 1024 entries) because each data entry is associated with a comparator logic. In a high-speed router, where millions of entries would be needed, such a large CAM will be hard to build and will be extremely expensive. Second, CAM does not provide a mechanism to find an expired entry for replacement. Adding such a mechanism to CAM would significantly complicate its design.

6 Conclusion

A well-engineered flow table is essential for the provision of QoS-related router functions such as traffic regulation, policy routing, and usage-based service accounting at high speed. Targeting two different models (distributed and centralized) of a router design, we proposed a software-based design for the distributed model and a hardware-based design for the centralized model. The main contributions of this work can be summarized as follows:

- In the software-based design, we developed an amortized garbage collection (AGC) technique to solve the garbage collection problem caused by the expiration of flows. We developed an accurate stochastic modeling of the garbage ratio, which measures the effectiveness of AGC. The analytical results were further validated through extensive trace-driven simulation.

- In the hardware-based design, we developed a dynamic set-associative scheme that significantly reduces the amount of overflow (with and without relocation) caused by traditional set-associative scheme at little extra cost. We demonstrated this through both stochastic modeling and trace-driven simulation, and these two independent results agree well with each other. Such a reduction in overflow is very important because the overflows are expected to be absorbed by a software-based flow table, which operates at a speed that is many times slower. Its pipelined system design helps achieve a high throughput of 100+ Gbps.

Acknowledgments

We would like to thank our editor, Dr. Ran Libeskind-Hadas, for managing a highly expeditious review process. We also thank Dr. Ran Libeskind-Hadas, Dr. Ellen Zegura, and the anonymous reviewers for their insightful and constructive suggestions that help improve the quality and readability of this paper.

Biographies

Jun Xu is an Assistant Professor in the College of Computing at Georgia Institute of Technology. He received the B.S. degree in computer science from Illinois Institute of Technology in 1995 and a Ph.D. degree in computer and information science from The Ohio State University in 2000. His current research interests include computer and network security, discrete algorithms for high-speed networks, performance modeling and simulation, and network hardware design.

Mukesh Singhal is a Full Professor and Gartner Group Endowed Chair in Network Engineering in the Department of Computer Science at The University of Kentucky, Lexington. He received a Bachelor of Engineering degree in Electronics and Communication Engineering with high distinction from Indian Institute of Technology, Roorkee, India, in 1980 and a Ph.D. degree in Computer Science from University of Maryland, College Park, in May 1986. His current research interests include operating systems, database systems, distributed systems, performance modeling, mobile computing, computer networks, and computer security. He has published over 140 refereed articles in these areas. He has coauthored two books titled "Advanced Concepts in Operating Systems", McGraw-Hill, New York, 1994 and "Readings

in Distributed Computing Systems",IEEE Computer Society Press, 1993. He is a Fellow of IEEE. He is currently serving in the editorial board of "IEEE Trans. on Knowledge and Data Engineering" and "Computer Networks". From 1998 to 2001, he served as the Program Director of Operating Systems and Compilers program at National Science Foundation.

References

[1] R. Edell, N. Mckeown, and P. Varaiya, "Billing users and pricing for tcp," *IEEE JSAC*, vol. 13, no. 7, pp. 1162–1175, Sept. 1995.

[2] S. Shenker and J. Wroclawski, "General characterization parameters for integrated services network elements," Sept. 1997, RFC 2215.

[3] P. Gupta and N. Mckeown, "Packet classification on multiple fields," in *Proc. of ACM SIGCOMM'99*, Sept. 1999, pp. 147–160.

[4] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. of ACM SIGCOMM'98*, Sept. 1998, pp. 203–214.

[5] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *Proc. of ACM SIGCOMM'98*, Sept. 1998, pp. 191–202.

[6] Jun Xu, Mukesh Singhal, and Joanne Degroat, "A novel cache architecture to support layer-four packet classification at memory access speeds," in *Proc. of Infocom'2000*, Israel, Mar. 2000.

[7] H. Chan, H. Alnuweiri, and V. Leung, "A framework for optimizing the cost and performance of next-generation ip routers," *IEEE JSAC*, vol. 17, no. 6, pp. 1013–1029, June 1999.

[8] RAMBUS Inc., *RAMBUS Technology Overview*, Feb. 1999.

[9] Jun Xu and Mukesh Singhal, "Cost-effective flow table designs for high-speed internet routers: Architecture and performance evaluation," Tech. Rep., The Ohio State University, Nov. 1999.

[10] K. Claffy, *Internet Workload Characterization*, Ph.D. thesis, UC San Diego, June 1994.

[11] W. Leland et al., "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Transaction on Networking*, pp. 1–15, Feb. 1994.

[12] V. Paxson and S. Floyd, "Wide area traffic: The failure of poisson modeling," *IEEE/ACM Transactions on Networking*, Apr. 1995.

[13] A. Feldmann, A. Gilbert, and W. Willinger, "Data networks as cascades: Investigating the multifractal nature of internet wan traffic," in *Proc. of ACM SIGCOMM'98*, Sept. 1998, pp. 42–55.

[14] J. Cao, W. Cleveland, D. Lin, and D. Sun, "On the nonstationarity of Internet traffic," in *Proc. of ACM Sigmetrics'01*, Cambridge, Massachusetts, USA, June 2001, pp. 102–112.

[15] K. Thompson, G. Miller, and R. Wilder, "Wide-area internet traffic patterns and characteristics," *IEEE Network*, pp. 10–23, Nov. 1997.

[16] D. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, Reading, Massachusetts, 1975.

[17] R. Nelson, *Probability, Stochastic Processes, and Queueing Theory*, Springer-Verlag, NY, 1995.

[18] NLANR, "Fixwest backbone traces," *http://moat.nlanr.net/Traces*, 1998.

[19] D. Patterson and J. Hennessy, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 1996.

[20] J. Carter and M. Wegman, "Universal classes of hash functions," *J. of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[21] Cisco Inc., *NetFlow Services and Applications*, Aug. 1999, http://www.cisco.com.

[22] P. Wilson, "Uniprocessor garbage collection techniques," *submitted to ACM Computing Surveys*, 2001.

[23] A. Broder and A. Karlin, "Multilevel adaptive hashing," in *Proc. of First Symposium of Discrete Algorithms*, 1990, pp. 43–53.

[24] E. Goto, T. Ida, and T. Gunji, "Parallel hashing algorithms," *Information Processing Letters*, vol. 6, no. 1, Feb. 1977.

[25] K. Hiraki, K. Nishida, and T. Shimada, "Evaluation of associative memory using parallel chained hashing," *IEEE Trans. on Computers*, vol. 33, no. 9, pp. 851–855, Sept. 1984.

[26] P. McKenney, "High-speed event counting and classification using a dictionary hash technique," in *Proc. of ICPP'89*, 1989, pp. 71–75.

[27] M. Dietzfelbinger et al., "Dynamic perfect hashing: upper and lower bounds," *SIAM Journal on Computing*, vol. 23, no. 4, pp. 738–761, 1994.

[28] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Communications of ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[29] F. Bodin and A. Seznec, "Skewed associativity improves program performance and enhances predictability," *IEEE transactions on computers*, vol. 46, no. 5, pp. 530–544, May 1997.