Cost-Effective Flow Table Designs for High-Speed Internet Routers

Jun Xu and Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
{jun,singhal}@cis.ohio-state.edu

November 19, 1999

Abstract

Provision of QoS-related router functions such as traffic shaping, policy routing, and usage-based accounting requires that a flow table store 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 packets per second). Targeting two different models (centralized and distributed) of router design, we propose a hardware-based design to be implemented in the main forwarding engine of a router, which is suitable for the centralized model, and a software-based design to be implemented on individual line cards, which is suitable for the distributed model. The hardware-based design, adapted from the architecture of an N-way set-associative cache, employs a novel dynamic set-associative scheme to reduce the overflow ratio that traditional set-associative scheme will incur, by a decent percentage, and a pipelined design to achieve a throughput of 100+ Gbps. The software-based design, adapted from hash table data structure, employs a novel "purging when convenient" technique to solve the garbage collection problem caused by the expiration of flows. The performance evaluation results from both trace-driven simulation and statistical analysis demonstrate that both designs are cost-effective for their targeted router models.

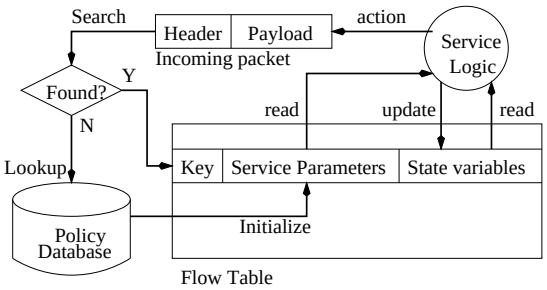

Fig.1(a) Operation Flowchart of a Flow Table

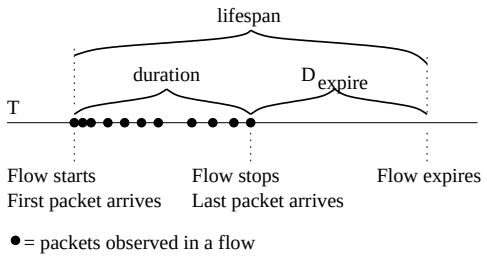

Fig. 1(b) Timeline of a Flow

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* [11], *traffic shaping* [22], and *policy routing* [13]. For traffic shaping, for each active flow, 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 to be applied to the packet, which are obtained from a relatively expensive *policy lookup* process [13, 16, 23], 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 more than 90 percent. The overview of a flow table is shown in Fig. 1(a). The schema of a flow table consists of *state variables* and *service parameters* used in service accounting (e.g., unit price), traffic conditioning (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 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, and a new entry needs to be created in the table for the flow. A *lookup* to the *policy database* is needed to determine the service parameters the flow entry should be initialized with. Though in this paper we will only discuss how the flow table supports the aforementioned router functions, it is clear from Fig. 1(a) that this flow table can also support many new flow-based network services that will appear in future Internet.

Similar to [10], we define a *flow* is a series of unidirectional packets that share the same <src_IP, dst_IP, src_port, dst_port, protocol> tuple, separated by no more than a tunable

timeout value D_{expire}, as shown in Fig. 1(b). 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 the flow stops is called the *duration* of the flow, during which the flow is *actually active*.

The flow table is a part of the forwarding engine: it serves as the *route cache* for the forwarding engine and implements a part of the forwarding logic. 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* [8]. In the centralized model, when a packet arrives at a line card, the packet header will be forwarded to a central forwarding engine through the *switching fabric*. The forwarding engine will decide the next-hop to forward the packet and its QoS if applicable, and send the decision back to the line card. On the other hand, in distributed router model, 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 [8]. These two router models place drastically different throughput requirements on the flow table. In the centralized model, one flow table will have to handle all the flow transactions of the router, while in the distributed router model, it only needs to handle the flow transactions at a line card.

Targeting two aforementioned router models, we proposed two cost-effective flow table designs: a hardware-based design (100+ Gbps throughput) for *centralized router model* and a software-based design (5 Gbps throughput) for *distributed router model*. 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 a novel *dynamic set-associative* scheme that reduces the overflow inherent with set-associativity by a decent percentage. Employing relatively inexpensive DRAM chips (800 MHz RAMBUS[21]), it nicely pipelines the system to "absorb" the DRAM access latency (60ns) and other system overhead so that a flow transaction can be completed almost every 20ns. As for the software-based design that employs hash table data structure, the major challenge comes from the need to purge the *expired flows* from the flow table. However,

garbage collection with hash table data structure in real time is not a trivial task. Our design employs a novel "purging when convenient" strategy that "absorbs" the overhead of garbage collection into that of probing and achieves a nice tradeoff between memory utilization and throughput. Great effort is spent to analyze, measure, and estimate every metric that may affect the functionality and performance of both designs.

At the time of writing, Cisco's NetFlow is the only implementation of flow table on routers. NetFlow supports up to 128K flows[9] and typically operates at a rate less than 1 Gbps. However, neither its underlying data structure (or architecture) nor its garbage collection technique is documented.

The paper is organized as follows. Section 2 discusses the router functions the flow table is going to support and related design issues. Section 3 and 4 present the architecture and performance evaluation of the hardware-based design and the software-based design, respectively. Section 5 concludes the paper.

2 Supported Router Functions and Related issues

In the following, we will briefly describe how the flow table supports the aforementioned QoS-related router functions. We focus only on the detail of each function that is relevant to our flow table design. The motivation and significance of each function, while interesting in its own right, is outside the scope of this paper.

Service Accounting: Usage-based service accounting was considered technologically expensive [11]. Now it can be performed in a 100+ Gbps router without bringing down its performance, with the help of a well-engineered flow table. Like in [3], the granule of our accounting scheme is a flow. When a flow is created, the unit price of each byte P (represented as 32-bit floating-point) in this flow will be determined and cached in the flow table as a service parameter. The factors that determine this unit price may include QoS, source/destination IP addresses [5], and the congestion level [15]. The flow table is responsible for bookkeeping two state variables, namely, the number of bytes C_{byte} (32-bit integer) and the number of packets C_{packet} (16-bit integer) in the flow. When a flow expires, its entry will be purged from the flow table and formatted into a *billing record*, which contains the layer-4 address, bytes/packet, total price, and start/end time of the flow, similar to a flow record exported by Cisco NetFlow [9].

Policy Routing: Policy routing determines the *next-hop* (or path ID such as Multi-protocol Label Switching (MPLS) label [6]), based on not only the destination IP, but also the source IP and port numbers as well [13]. This is referred to as layer-4 lookup [16, 23] in the literature. Recent advances [16, 23] boost its speed to 2 million lookups per second (MLPS), but it is still much slower[1] that 35+ MLPS needed to achieve 100+ Gbps throughput. Throughout this paper, we assume that the size of a packet is 350 bytes, which is a conservative estimate. Traffic statistics from all (about 10) MCI vBNS backbone nodes (http://www.vbns.net) shows that a *hit ratio* well above 95% can be achieved (D_{expire}=64s). With the flow table serving as route cache and with such a high hit ratio, one 2 MPLS classifier is enough to support a 100+ Gbps router.

Traffic Shaping: The traffic shaping algorithm under consideration here is the *token bucket* scheme [22] adopted by IETF, which can be implemented by the following *virtual scheduling algorithm* [12]. Suppose the *token accumulation rate* and *bucket capacity* for a flow is R and B, respectively. Each flow entry contains a state variable TS, which is set to CLK (the system clock) when the flow is created. When a packet of length L that belongs to the flow arrives, the system will test whether $(CLK + B/R) \geq (TS + L/R)$. If this is true, TS will be incremented by L/R and the packet is marked "IN" (conforming). Otherwise, the packet will be marked "OUT" (not conforming). Representing TS, B, and R as 32-bit floating-point numbers will achieve the necessary precision and range for such scheduling algorithms [24], but they would consume 12 bytes for each flow. To cut this space by half, we represent R and B each as a 8-bit index into an array of 256 possible values (stored in fast on-chip cache). To save computation, the rate array stores precomputed $1/R$ values instead of R values. It is shown in our technical report [1] that their 65536 combinations provide a very flexible set of supported traffic profiles. The computation by the service logic consists of the following four steps: (1) obtain $1/R$ and B from the corresponding array (2) compute $L * (1/R)$ and $B * (1/R)$ in parallel (3) compute $TS + L * (1/R)$ and $CLK + B * (1/R)$ in parallel (4) compare results from step 3. Since these are single-precision (not double-precision) floating point MUL/ADD operations (not DIV), all four steps can be completed within 20ns using

[1]Recursive Flow Classifier (RFC) proposed in [13] achieves 30 MLPS on enterprise firewall filters by exploiting their rich "structure and redundancy." However, it is not clear whether such a performance can be achieved on general layer-4 filters which can be much less structure-rich.

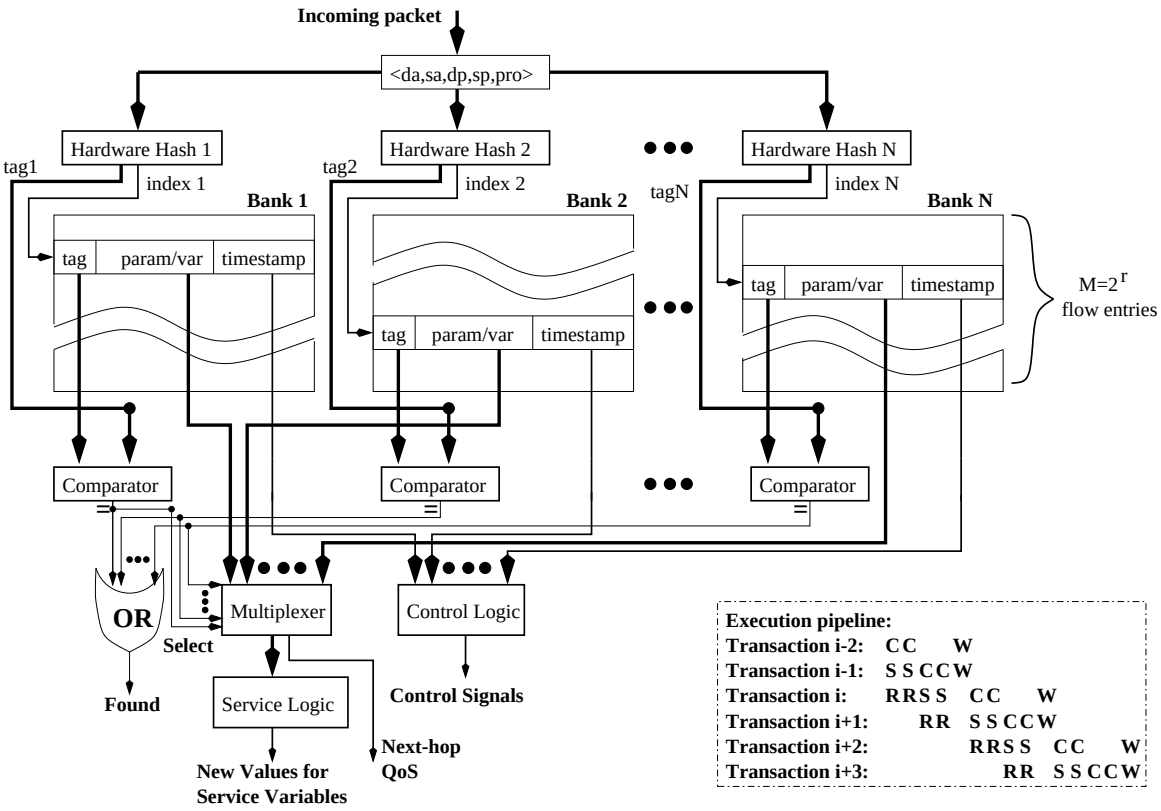

Figure 2: The architecture of the hardware-based design

commodity VLSI technology[2] (e.g., 0.25 micron).

To check if a flow entry has expired, we need to maintain a timestamp that records the time of the last access. We have shown in [1] that one byte is enough for this purpose. All the aforementioned state variables and service parameters consume 20 bytes. The *tag* (explained later) of the layer-4 address consumes 10 bytes. The control signals (one bit each) use another one byte. In this way, a flow entry fits into 32 bytes in the hardware-based design. In the software-based design, however, three more bytes are needed to store a pointer. We can squeeze 3 bytes out by discretizing the unit price of a flow into an array of 256 possible values, like what we did to R and B.

3 The Hardware-based Design

As shown in Fig. 2, 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

[2]The "wraparound" problem of CLK that may significantly complicate computation steps (3) and (4) can be eliminated by using two "ping-pong CLKs," as shown in our technical report[1].

a 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 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[3]. Each tag will be compared with the tag field contained in 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 flow needs to be brought into the flow table. If one or more of these N candidate entries (nodes) have expired (by checking the corresponding timestamp field), one of expired entries will be randomly[4] chosen for replacement. In case no entry has expired, an *overflow* happens. The overflowed flow entries will be "absorbed" into a software-based *assistant flow table*, the design of which is the topic of the next section. However, since the assistant flow table is about 20 times slower than the hardware-based design (called *master flow table*), it is desirable that the number of such overflow be as small as possible. We propose a dynamic set-associative scheme in which these N hash functions, h_1,h_2,..., h_N, are different and 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. It was shown in the literature that N hash functions randomly chosen from a class of hash functions H_3 satisfy the N-universal property [4, 7, 20]. We will describe H_3 and its hardware implementation in Section 3.3. Compared to the traditional set-associative scheme (we call it *static set-associative scheme*), where the N hash functions are identical, the dynamic set-associative scheme can reduce the number of overflow by a decent percentage, as will be shown in the next section.

3.1 Dynamic vs. Static Set-associative 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

[3]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.

[4]This can be perfectly emulated using a $2^N * (N + log(N))$ ROM and a $log(N)$-bit state vector[2].

relocation" vs. "without relocation"). In reality, relocation is always preferred because it reduces the amount of overflow in both dynamic scheme and static scheme, with little extra design complexity. However, statistical analysis of its overflow ratio is hard. On the other hand, the overflow of both static and dynamic schemes "without relocation" can be fairly 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.

Before we proceed with the analysis, we need to accurately define the metrics to be modeled. 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 *presumed active flows* AF. In the following discussion, unless the word "actual" is explicitly used, the term "active flows" refers to "presumed active flows." We define the *transaction overflow ratio* ($O_{transaction}$) as the number of flow transactions that is resolved by the assistant flow table divided by the total number of flow transactions. Let $\hat{NP}$ denote the number of packets in a flow. As a convention, every variable with a "hat" denotes a random variable, and the same variable without a "hat" will denote its *mean*. Let $\hat{NP}_{assistant}$ be the $\hat{NP}$ *conditioned* on the flow being in the assistant flow table. When the relocation is not allowed, $\hat{NP}_{assistant} = \hat{NP}$ because the "bad luck" (a *Bernoulli random variable*) of a flow being overflowed is statistically independent of its $\hat{NP}$. So $NP_{assistant}$=NP. Since the assistant flow table contains $AF * O_{entry}$ flows, they will handle $AF * O_{entry} * NP_{assistant}$ flow transactions. As the total number of flow transactions is $AF * NP$, $O_{transaction}$ is equal to $AF * O_{entry} * NP_{assistant}/(AF * NP)$ by definition. When NP and $NP_{assistant}$ cancel out, we get O_{entry}=$O_{transaction}$. This is not true when the relocation is allowed. Since the 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 $O_{entry} \geq O_{transaction}$. In the following analysis, when the relocation is not allowed, O is used to refer to both O_{entry} and $O_{transaction}$ because they are equal.

We define *load ratio* of the master flow table LR as the number of active flows AF divided by its total capacity $M * N$. It will be obvious that O is not only dependent on LR and

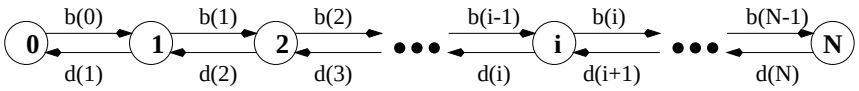

Figure 3. The Performance Modeling of the Dynamic Set-Associative Scheme

N, but also on its context. Our performance evaluation will be performed under a context called *dynamically-balanced flow system* (DBFS). In DBFS, every second NF flows expire and NF new flows arrive so that AF is stationary over the time. DBFS assumption is meaningful because it is shown in [10, 25] that AF tends to fluctuate little in a 5-minute interval, though it may change dramatically over a one-day period. Under such a 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. Also we assume *uniform hashing* throughout this paper. 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 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 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. Its dynamics can be modeled as a continuous Markov chain as shown in Fig. 3. 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. Here we assume that the arrivals of flows are independent of each other. When M is large, the arrival process is approximately Poisson. We assume that the departure process is also Poisson. 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 $\hat{S}$ denotes the number of active flows in this line. Then $N + 1$ unknowns,

	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 1: Overflow ratio comparison between static and dynamic

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 2: Statistics of the FIXWEST Traces Used in The Simulation

namely, $P[\hat{S} = i]$, i=0,1,...,N can be solved from the following $N+1$ linear 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 (to this line) is approximately Poisson, according to PASTA (Poisson Arrivals See Time Averages)[18], O is exactly $P[\hat{S} = N]$.

Table 1 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.

We conducted a simulation study to independently validate the theoretical results we have obtained and investigated the overflow ratio (O_{entry} and $O_{transaction}$) when relocation is allowed. Seven FIXWEST backbone traces[5] (see Table 2) were used in the simulation. Fig. 4 shows the simulation results 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. 3(a) shows the fluctuation of AF over the 1625s period. Fig. 3(b) compares the theoretical O with the experimental O in the dynamic scheme (without relocation). Fig. 3(c) compares the theoretical O with the experimental O in the static scheme (without relocation). In both Fig. 3(b) and Fig. 3(c), each point on the theoretical 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 error is about

[5]They are publicly available from http://moat.nlanr.net.

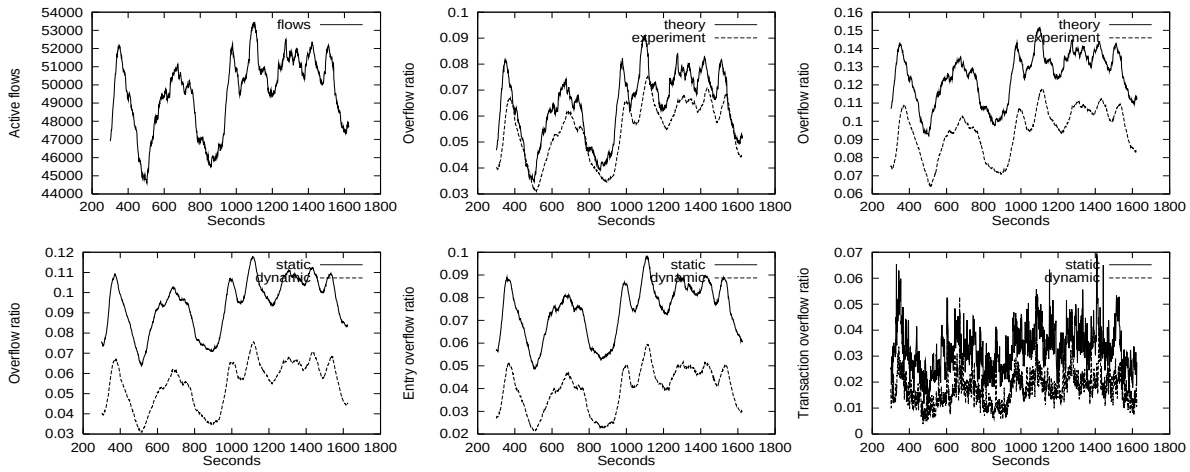

Figure 4: Dynamic vs. Static and Theory vs. Experiment (Top: (a)(b)(c), Bottom: (d)(e)(f))

15%, which can be attributed to the fluctuation of AF that makes the reality deviate a little bit from the DBFS context. Fig. 3(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 our analytical results in Table 1. Fig. 3(e) and Fig. 3(f) show experimental results when the relocation is allowed. Fig. 3(e) compares O_{entry} under dynamic scheme with O_{entry} under static scheme. Fig. 3(f) compares $O_{transaction}$ under dynamic scheme (lower cloud) with $O_{transaction}$ under static scheme (higher cloud). Fig. 3(e) and 3(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. 3(e) and 3(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% percent reduction in $O_{transaction}$ makes a difference because the assistant flow table is typically 20 times slower, as will be shown in Section 4.

Work proposed for a different application in [4] bears some similarity to our dynamic set-associative scheme. Let us explain it in our context and terms. It is interested in the overflow when the load suddenly increases from 0 to LR, which we refer to as the "wartime behavior." In contrast, we are interested in the "peacetime behavior" under the DBFS context. The performance modeling techniques (and results) of these two behaviors are totally different. Also, it operates in "prioritized mode" (lower-index bank has higher priority

of being replaced) while our dynamic set-associative scheme operates in "random mode" [2]. Dynamic set-associative scheme has also been employed in another work of ours [2] for a different application. But that work is interested in the "wartime behavior" under "random mode," the performance modeling technique of which is quite different from that of this paper ("peacetime"). Also, due to different system requirements, throughput optimization technique used in this paper is very different from the technique used in [2].

3.2 Throughput of the scheme

To achieve 100+ Gbps throughput, the architecture employs a new type of DRAM called RAMBUS [21]. After an *access latency* of 60ns (including the issuance of read/write commands), commercially available 800 MHz RAMBUS can deliver 16 bytes (the minimum *burst size*) every 10ns in *burst mode* (a series of accesses to consecutive memory locations). So it takes 20ns to *read* a 32-byte flow entry from the memory. As the fields that need to be updated occupy less than 16 bytes in the flow entry, they can be put together into the first 16 bytes. So for each flow transaction that results in a hit, we only need a "half write" (16 bytes), which takes 10ns. In our pipelined design (shown next), the *access latency* can be "hidden" by using FIFO buffering because RAMBUS allows a new read/write command to be issued immediately after the previous read/write command. Therefore, our concept of *read* and *write* only includes the *data transfer* and does not include the *access latency*.

The pipelining schedule is shown in Fig. 2. Each letter (R,S,C,W) is a 10 ns clock cycle. R and W stands for read and write, respectively. S is the search ("comparator," "multiplexer," and "OR" in Fig. 2) and replacement ("control logic") logic. S logic is quite simple and can be completed well within 20 ns using 0.25 μm CMOS VLSI. C stands for the computation in the service logic. We have explained in the last section that the service logic can be completed within 20 ns. Since each W only occurs to one memory bank, we can use *write merge* to improve the system throughput. As shown in Fig. 2, W in transaction i and W in transaction $i+1$ are executed simultaneously[6] so that two flow transactions are completed every 50 ns. 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.

We are still missing an important detail here. The W step for a flow transaction that

[6]Merging three or more writes into a cycle can further improve the throughput, as shown in [1].

causes a miss but not an overflow in the master flow table is to actually write an "interim record" (to avoid write-write *race condition*) into the entry chosen for replacement. When the result of a policy lookup becomes available or when the flow entry relocates back to the master flow table, another W is needed. This W is a "full write" of 32 bytes, which takes 20ns. We can merge it with a "half write" cycle so that it most probably only causes an additional "stall" of 10 ns, and cause a 20 ns stall with probability less than $2/N$. Everything considered, when N=6 and the miss ratio is 10%, the throughput of the architecture is 37.5 million transactions per second, which is equivalent to 106 Gbps when the average packet size is 350 bytes.

3.3 Implementation and Cost Analysis

A hash function in H_3 class 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:

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

Here, Q is a $r \times w$ matrix defined over GF(2)={0,1}. Each hash function in H_3 is uniquely corresponding to such a Q. The multiplication and addition in GF(2) is 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 the probability for r vectors randomly chosen from $\{0,1\}^w$ being *linearly independent* is $\prod_{i=0}^{r-1}(1-\frac{2^i}{2^w}) \geq 1-\sum_{i=0}^{r-1}\frac{2^i}{2^w} \geq 1-2^{r-w}$, $(1-\frac{2^i}{2^w})$ being the probability that v_i does not belong to the vector space spanned by $\{v_1, v_2, \cdots, v_{i-1}\}$. (The first inequality is a special case of Bernoulli's inequality.) In our system design, where $w = 97$ and r is typically between 20 and 24, this probability is greater than 0.999999. As they are linearly independent, 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.

H_3 hash function is very amenable to hardware implementation [20]. 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, which costs about 5 dollars (including packaging) assuming reasonable level of mass production (e.g., 3000+) [17]. This again justifies our dynamic set-associative scheme: the only difference in implementation between the dynamic scheme and the static scheme is $N-1$ (dynamic scheme needs N and static scheme needs 1) hash functions, which cost less than 50 dollars. This is well compensated by the amount of memory saved from the assistant table due to much smaller O_{entry}. We estimated that the total cost for all the logic chips used in the system is less than $250.

The major cost of the flow table comes from the memory, the cost of which is generally in proportion to its size. To estimate the amount of memory the system needs, we start by estimating the total number of active flows in a 100+ Gbps router. We introduce a metric called *normalized multiplexing level* (NML), which is the number of active flows per Mbps of traffic. We showed in [2] that NML is not going to increase in the future Internet where the traffic volume is much higher, because NML is in reverse proportion to the average bandwidth an Internet user enjoys. Based on this observation and recent traffic statistics from MCI vBNS backbone nodes (http://www.vbns.com/), we estimated that less than 16 million active flows will coexist in a 100+ Gbps router (with D_{expire} set to 60s), which will occupy 512 MB (each record is 32 bytes). However, since the "comfortable load" of the system is about 70%, the "comfortable" memory size is about 730 MB for such a router. However, as the price of DRAM continues to drop, this much of DRAM is expected to cost no more than $1000 in the near future. Such a cost is justified for a 100+ Gbps router.

4 Software-based Design

Since the software-based design handles the flow transactions at a single line card, it typically only needs to support up to 5 Gbps throughput. Using 800 MHz RAMBUS, we show that software implementation based on hash table data structure is able to achieve such a bandwidth provided we solve the problem of *garbage collection*. When a flow expires, the hash node that contains the flow entry becomes "garbage." Without a "garbage collection" mechanism[7], the system will eventually run out of memory. We propose a "purging when convenient" (PWC) scheme that "absorbs" the overhead of purging in *probing*. We introduce

[7] The hardware-based design implicitly solves this problem by replacing expired flows with new flows.

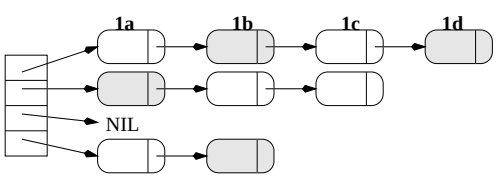

Fig. 5(a) Chaining with head table

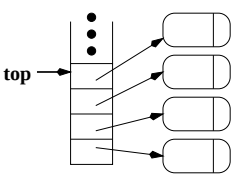

Fig. 5(b) Stack of free nodes

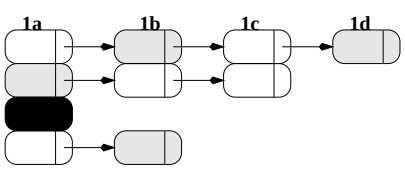

Fig. 5(c) Chaining without head table

a metric called *garbage ratio* that measures the effectiveness of the scheme, which can be accurately modeled using statistical techniques.

Here we consider *hashing with chaining*[8] as shown in Fig. 5(a). We assume that an H_3 hardware hash function is used, which transforms a layer-4 address into an index and a tag. The index is used to locate *a slot* in the *head table* that is pointing 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. In Fig. 5(a), the white nodes stand for active flows and the gray nodes stand for expired flows. In the following discussion, we use the word "node" and "flow" interchangeably.

Our PWC scheme works as follows. While the probe is carried out along the linked list, if the current node has expired, it will be deleted from the linked list and returned to a stack of *free nodes*, as shown in Fig. 5(b). For example, in Fig. 5(a), suppose node 1c is the flow that is searched for. Then, when the probe is carried out in the top linked list, it finds that node 1b has expired and will delete it from the chain. However, node 1d will not be affected because it is behind node 1c. So, the purging occurs only when it is "convenient." The effectiveness of the scheme is measured by *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 exactly the "gray/white" ratio in Fig. 5(a). The garbage ratio is analyzed under DBFS context as follows.

Under DBFS, NF flows arrive and NF flows expire each second so that AF remains constant. Let $T_{lifespan}$ be the average lifespan of a flow (it is actually the average lifespan of a white node). Then according to *Little's law*[18], $AF = T_{lifespan} * NF$. Let $\hat{T}_{purge}$ be the latency between the time a node expires and the time it is purged by PWC, 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 $GR = \frac{T_{purge}*NF}{T_{lifespan}*NF} = \frac{T_{purge}}{T_{lifespan}}$. Let $T_{duration}$ be the average duration of a flow, duration which the flow is *actually active*. By a similar argument (using Little's law), the ratio of the *presumed*

[8] We do not use *open hashing* algorithms because it makes garbage collection a complicated task[14].

active flows that are *actually active* is $\frac{T_{lifespan}-D_{expire}}{T_{lifespan}}$ (Note $T_{lifespan} = T_{duration} + D_{expire}$). We call this ratio the *actually active ratio*, and denote it as ρ.

Now the task is to analyze $\hat{T}_{purge}$. 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. When M is large, the number of active nodes (denoted as $\hat{U}$) that map to the slot follow approximately the Poisson distribution $P[\hat{U} = i]$=$exp(-LF) * LF^i/(i!)$. So among M slots of the hash table, approximately $M * P[\hat{U} = i]$ of them contain i active nodes, i=0,1,2,... So 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: $P[\hat{V} = i]$=$\frac{i*M*P[\hat{U}=i]}{AF}$= $\frac{i*exp(-LF)*LF^i}{LF*(i!)}$, i=1,2,... This is derived using informal analysis, but one can show 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:

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

Now we are ready to analyze $\hat{T}_{purge}$. Suppose exactly at time 0, a node ND expires. Just before time 0, ND can be any of the AF nodes with equal probability $1/AF$. Suppose the slot that ND belongs to have i nodes and ND is the k_{th} along the probing sequence (Equation (2) tells us the probability of such an event). Suppose among the $i-k$ presumed 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 way is that one among those $\hat{X}$ actually active nodes get a *visitor* (an incoming packet that belongs to the flow). The second way is that 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 $T_{lifespan}/NP$. 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. 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 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 get

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

$T_{lifespan}$ can be derived from ρ and the tunable parameter D_{expire}. So GR is a function of three independent parameters I, ρ, and LF. An analysis of the (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 from (4) 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 [14]. $LF = 4$ strikes a nice tradeoff between memory and speed. When $LF = 4$, GR is about 0.16, and the *transaction processing time* can be controlled under 560ns (just fast enough to support 5 Gbps).

I and ρ are traffic-dependent parameters, but fortunately GR is not sensitive to the changes of these two parameters. Otherwise, GR will be heavily 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. So the result of (4) can be applied to Internet traffic in general. 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 from the results (0.89, 0.40 and 0.16) obtained using (4).

We conducted a simulation study to independently validate (4) using FIXWEST traces (see Table 2), and the results are presented in Fig. 6. Fig. 6(a), 6(b), and 6(c) show AF in FIXWEST trace 1,2, and 4, respectively (D_{expire} is set to 60s, 30s, and 40s, respectively). The number of slots M is 16384. Fig. 4(d)(e)(f) compare the theoretical GR with the

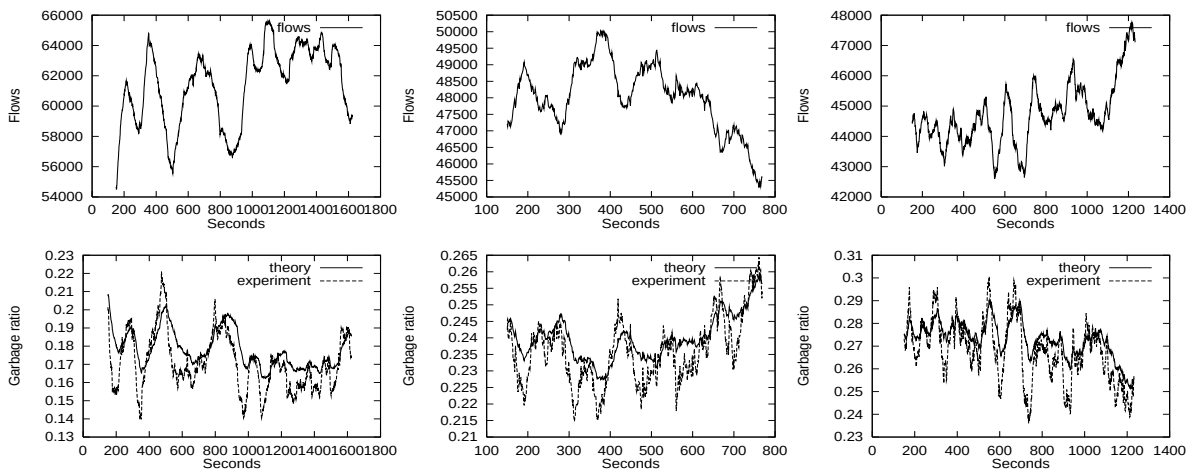

Figure 6: Theory vs. Experiment (Top: (a)(b)(c), Bottom: (d)(e)(f))

experimental GR in these three traces. Each point on the theoretical 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 using our simulation program. In all three cases, the error is about 5 percent. This shows that our modeling of GR very well matches the reality.

Implicitly, the existence of the *head table* (see Fig. 5(a)) makes our discussion much easier. However, it does not make sense in the real implementation. It adds one more memory access to the probe length and wastes about 10% of the memory (each pointer is 3 byte long). Now it is the time to remove it. Fig. 5(c) shows the aftermath of the removal of the head table from the hash table shown in Fig. 5(a). The black node in Fig. 5(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. 5(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. 5(b)) adds an additional 10 percent overhead (to store pointers). So the total 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.

The throughput of the software-based design is analyzed as follows. We assume that the system consists of a 200 MHz DLX [19] CPU (now considered low-end) with decent amount of on-chip cache and typical pipelining support, a RAMBUS chip, and reasonable hardware support such as timers. We assume that the PWC (purging when convenient)

code can fit into on-chip cache because it is very small (less than 2k). When $LF = 4$, the average processing time of a flow transaction triggered by an arbitrary packet Z is analyzed as follows. Let MR be the miss ratio, the percentage of the flow transactions that cause a *miss*. If Z belongs to an active node (we call it *Z's flow*) in the hash table, which happens with probability $1 - MR$, it takes an average of $1 + LF/2$ memory accesses to find the node. Unlike in the hardware-based design, the read command to access the next node in the linked list can not start until the read command to access the current node finishes due to data dependence. So each memory access takes 60+20+10=90 ns. 60ns is the access latency, 20ns is the transfer time of 32 bytes, and the other 10 ns is the time to prepare for the next access (2 DLX instructions). This takes an average of 270ns to find the node when $LF = 4$. During the probing, if it turns out that the current node is not Z's flow, it takes 13 instructions to perform the *routine inspection*, that is, to check if the current node expires and to check if it is Z's flow (we don't know until we check). They can be finished within 15 clock cycles (75ns), counting in two pipeline stalls caused by data dependence. So they can be executed in parallel with the prefetching (takes 80ns) of the next node. So, the overhead of routine inspection is completely absorbed into the memory access overhead. When the probe reaches Z's node, 44 instructions are needed for the routine inspection (this one is not absorbed) and service logic, which can be finished in 48 clock cycles (240ns). Another 20ns is needed to write the result back to the node. We don't count the 60ns latency here because the next flow transaction most probably (if there is no race condition) can start as soon as the write command is issued. So the total access time is 530ns when Z's node is found. In a similar way, we analyzed that the total access time when Z's node is not found is less than 700ns. This counts in the unsuccessful probe, deletion of expired nodes, free node stack operation, and writing of "interim" and "final" records. Therefore, when $MR = 0.9$, the average processing time of a flow transaction is 700*0.1+530*0.9=547ns, resulting in a bandwidth of 5.23 Gbps (average packet size is 350 bytes).

Finally, we would like to discuss why PWC is preferred over two alternative schemes. One is "proactive purging" in which each probe goes "out of way" to inspect all the nodes in the slot. The GR of this scheme can be accurately estimated by replace three occurrences of (i-k) with i in equation (4). We found that it increases the probe length significantly (from $1 + LF/2$ to $1 + LF$) but saves only a little more memory (subtract about 5 percent from GR). The other alternative is "purging when broke" in which the purging is performed (by

traversing all nodes in the hash table sequentially) only when there is no free node. Our analysis shows that it does not achieve a better memory and speed tradeoff than PWC [1] Garbage collection heuristics similar to PWC might have been proposed in unrelated areas like web caching. However, to the best of our knowledge, no work similar to our modeling of GR in PWC has been reported in literature of any kind.

5 Conclusion

We showed that a well-engineered flow table is essential for the provision of QoS-related router functions such as traffic shaping, policy routing, and usage-based service at high speed, and discussed related design issues. Targeting two different models (centralized and distributed) of router design, we proposed a hardware-based design, adapted from N-way set-associative scheme, for the centralized model, and a software-based design, based on hash table data structure, for the distributed model. In the hardware-based design, we developed a novel dynamic set-associative scheme that significantly reduces the amount of overflow (with and without relocation) caused by traditional set-associative scheme at no extra cost, as demonstrated by both theoretical and simulation results. Its pipelined system design helps achieve a high throughput over 100 Gbps. In the software-based design, we developed a "purging when convenient" (PWC) technique to solve the garbage collection problem caused by the expiration of flows. Results from our statistical modeling of the garbage ratio, which measures the effectiveness of PWC, very well matches the results of trace-driven simulation. We showed how the system achieves 5 Gbps using software on a low-end CPU by overlapping memory access with program execution.

References

[1] A. Anonymous. Cost-effective flow table designs for high-speed internet routers: Architecture and performance evaluation. Technical report, Instituion Anonymous, Oct. 1999.

[2] A. Anonymous. Title anonymous. In *Proc. of Infocom'2000*, Israel, Mar. 2000.

[3] H. Braun, K. Claffy, and G. Polyzos. A framework for flow-based accounting on the internet. In *Proc. of the Singapore Intl Conference on Networks*, 1993.

[4] A. Broder and A. Karlin. Multilevel adaptive hashing. In *Proc. of First Symposium of Discrete Algorithms*, pages 43–53, 1990.

[5] N. Brownlee. New zealand experiences with network traffic charging. *ConneXions*, 8(11):12–19, Nov. 1994.

[6] R. Callon et al. A framework for multiprotocol label switching, July 1999. Internet Draft.

[7] J. Carter and M. Wegman. Universal classes of hash functions. *J. of Computer and System Sciences*, 18(2):143–154, 1979.

[8] H. Chan, H. Alnuweiri, and V. Leung. A framework for optimizing the cost and performance of next-generation ip routers. *IEEE JSAC*, 17(6):1013–1029, June 1999.

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

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

[11] R. Edell, N. Mckeown, and P. Varaiya. Billing users and pricing for tcp. *IEEE JSAC*, 13(7):1162–1175, Sept. 1995.

[12] D. Ginsburg. *ATM: Solutions for Enterprise Internetworking*. Addison Wesley, 1996.

[13] P. Gupta and N. Mckeown. Packet classification on multiple fields. In *Proc. of ACM SIGCOMM'99*, pages 147–160, Sept. 1999.

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

[15] Y. Korilis, T. Varvarigou, and S. Ahuja. Incentive-compatible pricing strategies in noncooperative networks. In *Proc. of Infocom'98*, 1998.

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

[17] S. I. A. T. Needs. *The National Technology Roadmap for Semiconductors*. Semiconductor Industry Association, 1997.

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

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

[20] M. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Trans. on Computers*, 46(12):1378–1381, Dec. 1997.

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

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

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

[24] D. Stephens, J. Bennett, and H. Zhang. Implementing scheduling algorithms in high-speed networks. *IEEE JSAC*, 17(6):1145–1158, June 1999.

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