Chapter 9: Designing Statistics Counters from Slower Memories

Apr 2008, Sonoma, CA

Contents

List of Dependencies

- **Background:** The memory access time problem for routers is described in Chapter 1. Section 1.5.3 describes the use of caching techniques to alleviate memory access time problems for routers in general.

Additional Readings

- **Related Chapters:** The caching hierarchy described in this chapter was first described to implement high-speed packet buffers in Chapter 7. A similar technique is also used in Chapter 8 to implement high-speed packet schedulers.

Table: *List of Symbols.*

T	Time Slot
T_{RC}	Random Cycle Time of Memory
M	Total Counter Width
m	Counter Width in Cache
P	Minimum Packet Size
R	Line Rate
N	Number of Counters
$C(i,t)$	Value of Counter i at Time t
b	Number of Time Slots between Updates to DRAM

Table: *List of Abbreviations.*

LCF	Longest Counter First
CMA	Counter Management Algorithm
SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory

"There are three types of lies - lies, damn lies, and statistics".

— Benjamin Disraeli[†]

9

Designing Statistics Counters from Slower Memories

9.1 Introduction

Many applications on routers maintain statistics. These include firewalling [194] (especially stateful firewalling), intrusion detection, performance monitoring (*e.g.*, RMON [195]), network tracing, Netflow [196, 197, 198], server load balancing, and traffic engineering [199] (*e.g.*, policing and shaping). In addition, most routers maintain statistics to facilitate network management.

✎**Example 9.1.** Figure 9.1 shows four examples of measurement counters in a network — (a) a storage gateway keeps counters to measure disk usage statistics, (b) the central campus router maintains measurements of users who attempt to maliciously set up connections or access forbidden data, (c) a local bridge keeps usage statistics and performs load balancing and directs connections to the least-loaded web-server, and (d) a gateway router that provides connectivity to the Internet measures customer traffic usage for billing purposes.

[†]Also variously attributed to Alfred Marshall and Mark Twain.

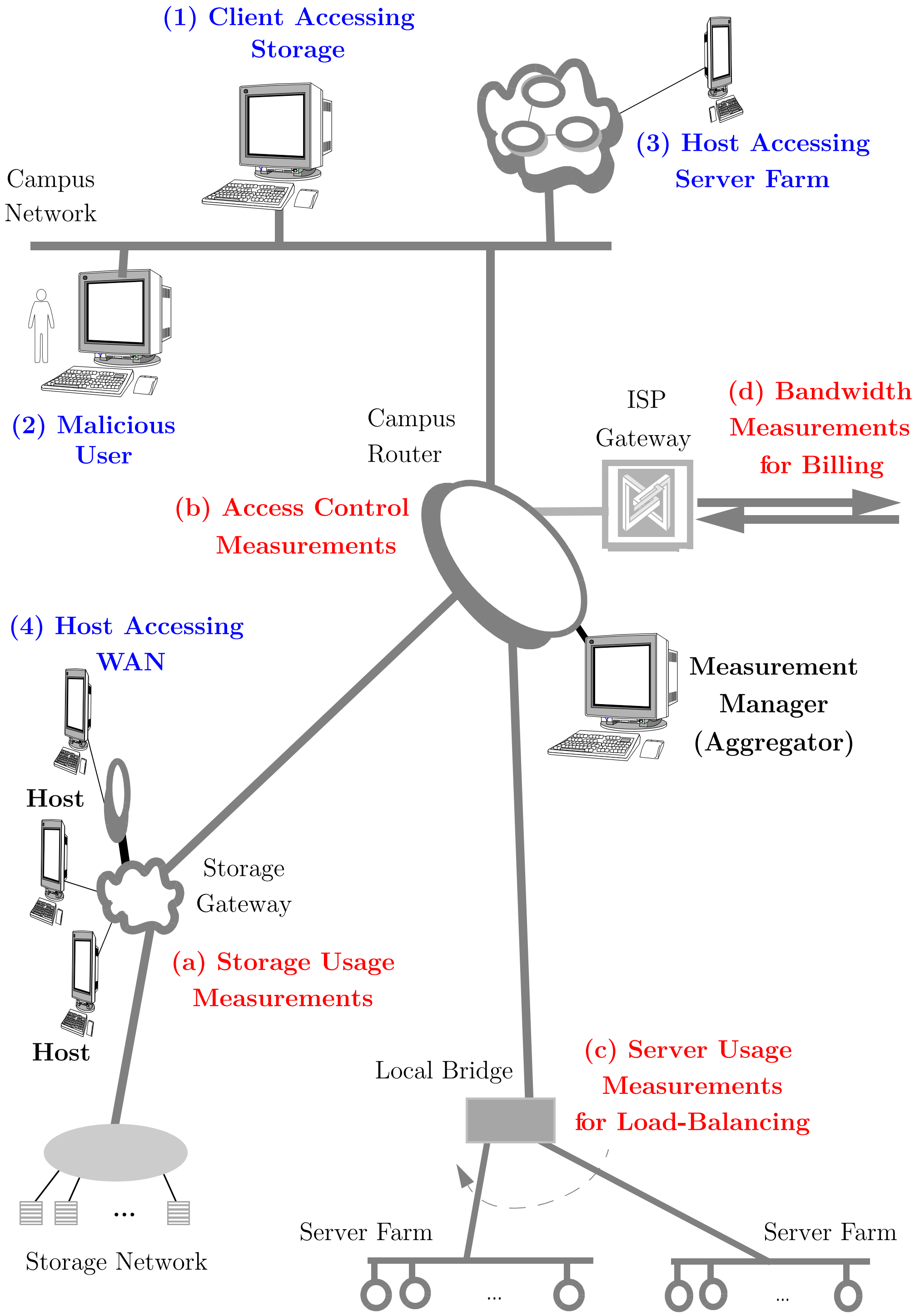

Figure 9.1: *Measurement infrastructure.*

The general problem of statistics maintenance can be characterized as follows: When a packet arrives, it is first classified to determine which actions will be performed on the packet — for example, whether the packet should be accepted or dropped, whether it should receive expedited service or not, which port it should be forwarded to, and so on. Depending on the chosen action, some statistics counters are updated.

We are not concerned with how the action to be performed is identified. The task of identification of these actions is usually algorithmic in nature. Depending on the application, many different techniques have been proposed to solve this problem (*e.g.*, address lookup [134], packet classification [184], packet buffering [173, 174], QoS scheduling [186], *etc.*).

We are concerned with the statistics that these applications measure, once the action is performed and the counter to be updated (corresponding to that action) is identified. The statistics we are interested in here are those that count events: for example, the number of fragmented packets, the number of dropped packets, the total number of packets arrived, the total number of bytes forwarded to a specific destination, *etc.* In the rest of this chapter we will refer to these as *counters.*

As we will see, almost all Ethernet switches and Internet routers maintain statistics counters. At high speeds, the rate at which these counters are updated causes a memory bottleneck. It is our goal to study and quantitatively analyze the problem of maintaining these counters,[1] and we are motivated by the following question — *How can we build high-speed measurement counters for routers and switches, particularly when statistics need to be updated faster than the rate at which memory can be accessed?*

9.1.1 Characteristics of Measurement Applications

Note that if we only want to keep measurements for a particular application, we could exploit the characteristics of that application to build a tailored solution. For example, in [201], Estan and Varghese exploit the heavy-tailed nature of traffic to quickly

[1] Our results were first published in [200], and we were not aware of any previous work that describes the problem of maintaining a large number of statistics counters.

identify the largest flows and count them (instead of counting all packets and flows that arrive) for the Netflow application [196, 197].

☞**Observation 9.1.** Most high-speed routers offer limited capability to count packets at extremely high speeds. Cisco Systems' Netflow [196] can sample flows, but the sampling only captures a fraction of the traffic arriving at line rate. Similarly, Juniper Networks [197] can filter a limited set of flows and maintain statistics on them.

But we would like a general solution that caters to a broad class of applications that have the following characteristics:

1. Our applications maintain a large number of counters. For example, a routing table that keeps a count of how many times each prefix is used, or a router that keeps a count of packets belonging to each TCP connection. Both examples would require several hundreds of thousands or even millions of counters to be maintained simultaneously, making it infeasible (or at least very costly) to store them in SRAM. Instead, it becomes necessary to store the counters in off-chip, relatively slow DRAM.
2. Our applications update their counters frequently. For example, a 100 Gb/s link in which multiple counters are updated upon each packet arrival. These read-modify-write operations must be conducted at the same rate as packets arrive.
3. Our applications mandate that the counter(s) be correctly updated every time a packet arrives; no packet must be left unaccounted for in our applications.
4. Our applications require measurement at line rates without making any assumptions about the characteristics of the traffic that contribute to the statistics. Indeed, similar to the assumptions made in Chapter 7, we will require hard performance guarantees to ensure that we can maintain counters for these applications at line rates, even in the presence of an adversary.

The only assumption we make is that these applications do not need to read the value of counters in every time slot.[2] They are only interested in updating their values. From the viewpoint of the application, the counter update operation can be performed in the background. Over time, a control path processor reads the values of these counters (typically once in a few minutes on most high-speed routers) for post processing and statistics collection.

9.1.2 Problem Statement

If each counter is M bits wide, then a counter update operation is as follows: 1) read the M bit value stored in the counter, 2) increment the M bit value, and 3) write the updated M bit value back. If packets arrive at a rate R Gb/s, the minimum packet size is P bits, and if we update C counters each time a packet arrives, the memory may need to be accessed (read or written) every $P/2CR$ nanoseconds.

✎**Example 9.2.** Let's consider the example of 40- byte TCP packets arriving on a 40 Gb/s link, each leading to the updating of two counters. The memory needs to be accessed every 2 ns, about 25 times faster than the random-access speed of commercial DRAMs today.

If we do an update operation every time a packet arrives, and update C counters per packet, then the minimum bandwidth R_D required on the memory interface where the counters are stored would be at least $2RMC/P$. Again, this can become unmanageable as the size of the counters and the line rates increase.

9.1.3 Approach

In this chapter, we propose an approach similar to the caching hierarchy introduced in Chapter 7, which uses DRAMs to maintain statistics counters and a small, fixed amount of (possibly on-chip) SRAM. We assume that N counters of width M bits are to be stored in the DRAM, and that N counters of width $m \ll M$ bits are stored in

[2]There are applications for which the value of the counter is required to make data-path decisions. We describe memory load balancing techniques to support these applications in Chapter 10.

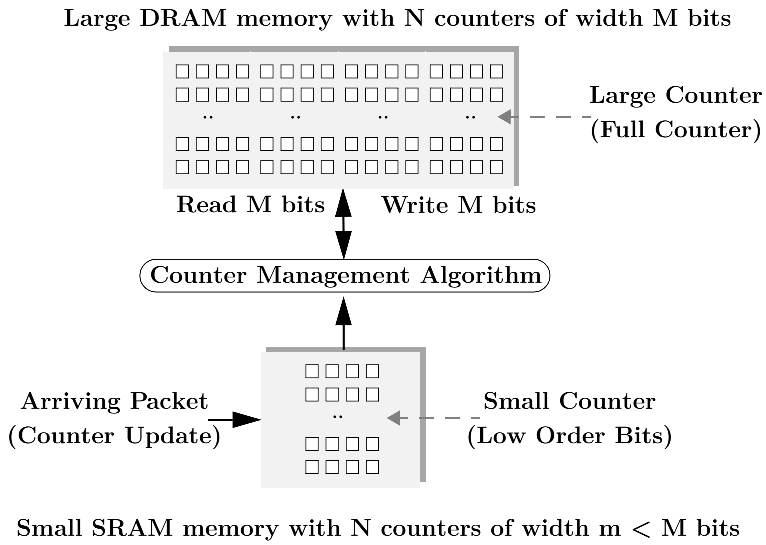

Figure 9.2: *Memory hierarchy for the statistics counters. A fixed-sized ingress SRAM stores the small counters, which are periodically transferred to the large counters in DRAM. The counters are updated only once every b time slots in DRAM.*

SRAM. The counters in SRAM keep track of the number of updates not yet reflected in the DRAM counters. Periodically, under the control of a counter management algorithm (CMA), the DRAM counters are updated by adding to them the values in the SRAM counters, as shown in Figure 9.2. The basic idea is as follows:

> **Idea.** *"If we can aggregate and update the DRAM counters relatively infrequently (while allowing for frequent on-chip SRAM updates), the memory bandwidth requirements can be significantly reduced".*

We are interested in deriving strict bounds on the size of the SRAM such that – irrespective of the arriving traffic pattern – none of the counters in the SRAM overflow, or the access rate and bandwidth requirements on the DRAM are decreased, while still ensuring correct operation of the counters.

We will see that the size of the SRAM, and the access rate of the DRAM, both depend on the CMA used. The main result of this chapter is that there exists a CMA (which we call largest counter first, LCF) that minimizes the size of the SRAM. We derive necessary and sufficient conditions on the sizes of the counters (and hence of the SRAM that stores all these counters), and we prove that LCF is optimal. An example of how this technique can be used is illustrated below.

✎**Example 9.3.** Consider a 100 Gb/s line card on a router that maintains a million counters. Assume that the maximum size of a counter is 64 bits and that each arriving packet updates one counter. Our results indicate that a statistics counter can be built with a commodity DRAM [3] with access time 51.2 ns, a DRAM memory bandwidth of 1.25 Gb/s, and 9 Mb of SRAM.

9.2 Caching Memory Hierarchy

We will now describe the memory hierarchy used to hold the statistics counters, and in the sections that follow we will describe the LCF-CMA (Largest Counter First Counter Management Algorithm).

ℵ**Definition 9.1. Minimum Packet Size, P:** Packets arriving at a switch have variable lengths. We will denote by P the minimum length that a packet can have.

As mentioned in Chapter 1, we will choose to re-define the unit of time as necessary to make our analysis simpler to understand. In what follows, we will denote a time slot as the time taken to receive a minimum-sized packet at a link rate R. The SRAM is organized as a statically allocated memory, consisting of separate storage space for each of the N counters. We will assume from this point forward that an arriving packet increments only one counter. If instead we wish to consider the case where C counters are updated per packet, we can consider the line rate on the interface to be CR.

Each counter is represented by a large counter of size M bits in the DRAM, and a small counter of size $m < M$ bits in SRAM. The small counter counts the most recent events, while the large counter counts events since the large counter was last updated. At any time instant, the correct counter value is the sum of the small and large counters.

Updating a DRAM counter requires a read-modify-write operation: 1) Read an M bit value from the large counter, 2) Add the m bit value of the corresponding small counter to the large counter, 3) Write the new M bit value of the large counter to DRAM, and 4) Reset the small counter value.

In this chapter, we will assume that the DRAM access rate is slower than the rate at which counters are updated in SRAM. And so, we will update the DRAM only once every b time slots, where $b > 1$ is a variable whose value will be derived later. So the I/O bandwidth required from the DRAM is $R_D = 2RM/Pb$, and the DRAM is accessed only once every $A_t = Pb/2R$ time slots to perform a read or a write operation. Thus, the CMA will update a large counter only once every b time slots. We will determine the minimum size of the SRAM as a function $g(.)$ and show that it is dependent on N, M, and b. Thus, the system designer is given a choice of trading off the SRAM size $g(N, M, b)$ with the DRAM bandwidth R_D and access time A_t.[3]

ℵ**Definition 9.2. Count** $C(i,t)$**:** At time t, the number of times that the i^{th} small counter has been incremented since the i^{th} large counter was last updated.

ℵ**Definition 9.3. Empty Counter:** A counter i is said to be empty at time t if $C(i,t) = 0$.

We note that the correct value of a large counter may be lost if the small counter is not added to the large counter in time, *i.e.*, before an overflow of the small counter. Our goal is to find the smallest-sized counters in the SRAM, and a suitable CMA,

[3]The variable b ($b \geqslant 1$) is chosen by the system designer. If $b = 1$, no SRAM is required, but the DRAM must be fast enough for all counters to be updated in DRAM.

such that a small counter cannot overflow before its corresponding large counter is updated.[4]

9.3 Necessity Conditions on any CMA

Theorem 9.1. *(Necessity) Under any CMA, a counter can reach a count* $C(i,t)$ *of*

$$\frac{\ln\left[(b/(b-1))^{(b-1)}(N-1)\right]}{\ln(b/(b-1))}. \tag{9.1}$$

Proof. We will argue that we can create an arrival pattern for which, after some time, there exists k such that there will be $(N-1)/((b-1)/b)^k$ counters with count $k+1$ irrespective of the CMA.

Consider the following arrival pattern. In time slot $t = 1, 2, 3, \ldots, N$, small counter t is incremented. Every b^{th} time slot one of the large counters is updated, and the corresponding small counter is reset to 0. So at the end of time slot N, there are $N(b-1)/b$ counters with count 1, and N/b empty counters. During the next N/b time slots, the N/b empty counters are incremented once more, and N/b^2 of these counters are now used to update the large counter and reset. So we now have $[N(b-1)/b] + [N(b-1)/b^2]$ counters that have count 1. In a similar way, we can continue this process to make $N-1$ counters have a count of 1.

During the next time $N-1$ slots, all $N-1$ counters are incremented once, and $1/b$ of them are served and reset to zero. Now, assume that all of the remaining approximately N/b empty counters are incremented twice in the next $2N/b$ time slots,[5] while $2N/b^2$ counters become empty due to service. Note that the empty counters decreased to $2N/b^2$ from N/b (if $b=2$, there is no change). In this way, after some time, we can have $N-1$ counters of count 2.

[4]It is interesting to note that there is an analogy between the cache sizes for statistics counters and the buffer cache sizes that we introduced in Chapter 7. (See Box 9.1.)

[5]In reality, this is $(N-1)/b$ empty counters and $2(N-1)/b$ time slots.

By continuing this argument, we can arrange for all $N-1$ counters to have a count $b-1$. Let us denote by T the time slot at which this first happens.

During the interval from time slot $2(N-1)$ to $3(N-1)$, all of the counters are again incremented, and $1/b$ of them are served and reset to 0, while the rest have a count of two. In the next $N-1$ time slots, each of the counters with size 2 is incremented and again $1/b$ are served and reset to 0, while the rest have a count of three. Thus there are $(N-1)((b-1)/b)^2$ counters with a count of three. In a similar fashion, if only non-empty counters keep being incremented, after a while there will be $(N-1)((b-1)/b)^k$ counters with count $k+1$. Hence there will be one counter with count:

$$\begin{aligned}\frac{\ln(N-1)}{\ln(b/(b-1))} &= \frac{\ln(N-1)+(b-1)\ln(b/(b-1))}{\ln(b/(b-1))} \\ &= \frac{\ln\left[(b/(b-1))^{(b-1)}(N-1)\right]}{\ln(b/(b-1))}.\end{aligned}$$

Thus, there exists an arrival pattern for which a counter can reach a count $C(i,t)$ of

$$\frac{\ln\left[(b/(b-1))^{(b-1)}(N-1)\right]}{\ln(b/(b-1))}.$$ ❐

9.4 A CMA that Minimizes SRAM Size

9.4.1 LCF-CMA

We are now ready to describe a counter management algorithm (CMA) that will minimize the counter cache size. We call this, *longest counter first (LCF-CMA)*.

Algorithm 9.1: The longest counter first counter management algorithm.

```
input  : Counter Values in SRAM and DRAM.
output: The counter to be replenished.

/* Calculate counter to replenish */
repeat every b time slots
    CurrentCounters ← SRAM[(1, 2, ..., N)]
    /* Calculate longest counter */
    C_Max ← FindLongestCounter(CurrentCounters)
    /* Break ties at random */
    if C_Max > 0 then
        DRAM[C_Max] = DRAM[C_Max] + SRAM[C_Max]
        /* Set replenished counter to zero */
        SRAM[C_Max] = 0

/* Service update for a counter */
repeat every time slot
    if ∃ update for c then
        SRAM[c] = SRAM[c] + UpdateCounter(c )
```

Algorithm Description: Every b time slots, LCF-CMA selects the counter i that has the largest count. If multiple counters have the same count, LCF-CMA picks one arbitrarily. LCF-CMA updates the value of the corresponding counter i in the DRAM and sets $C(i, t) = 0$ in the SRAM. This is described in Algorithm 9.1.

9.4.2 Optimality

Theorem 9.2. *(Optimality of LCF-CMA) Under all arriving traffic patterns, LCF-CMA is optimal, in the sense that it minimizes the count of the counter required.*

Proof. We give a brief intuition of this proof here. Consider a traffic pattern from time t, which causes some counter C_i (which is smaller than the longest counter at time t) to reach a maximum threshold M^*. It is easy to see that a similar traffic pattern can cause the longest counter at this time t to exceed M^*. This implies that not serving the longest counter is sub-optimal. A detailed proof appears in Appendix I. ❒

9.4.3 Sufficiency Conditions on LCF-CMA Service Policy

Theorem 9.3. *(Sufficiency) Under the LCF-CMA policy, the count $C(i,t)$ of every counter is no more than*

$$\boxed{S \equiv \frac{\ln bN}{\ln(b/(b-1))}. \qquad (9.2)}$$

Proof. (By Induction) Let $d = b/(b-1)$. Let $N_i(t)$ denote the number of counters with count i at time t. We define the following Lyapunov function:

$$F(t) = \sum_{i \geqslant 1} d^i N_i(t). \tag{9.3}$$

We claim that under LCF policy, $F(t) \leqslant bN$ for every time t. We shall prove this by induction. At time $t = 0$, $F(t) = 0 \leqslant bN$. Assume that at time $t = bk$ for some k, $F(t) \leqslant bN$. For the next b time slots, some b counters with count $i_1 \geqslant i_2 \geqslant \cdots \geqslant i_b$ are incremented. Even though not required for the proof, we assume that the counter values are distinct for simplicity. After the counters are incremented, they have counts $i_1+1, i_2+1, \ldots, i_b+1$ respectively, and the largest counter among all the N counters is serviced. The largest counter has at least a value $C(.) \geqslant i_1 + 1$.

- ***Case 1*:** If all the counter values at time t were nonzero, then the contribution of these b counters in $F(t)$ was: $\alpha = d^{i_1} + d^{i_2} + \cdots + d^{i_b}$. After considering the values of these counters after they are incremented, their contribution to $F(t+b)$ becomes $d\alpha$. But a counter with a count $C(.) \geqslant i_1 + 1$ is served at time $t+b$ and its count becomes zero. Hence, the decrease to $F(t+b)$ is at least $d\alpha/b$. This is because the largest counter among the b counters is served, and the contribution of the largest of the b counters to the Lyapunov function must be at least greater than the average value of the contribution, $d\alpha/b$. Thus, the net increase is at most $d\alpha[1-(1/b)]-\alpha$. But $d[1-(1/b)]=1$. Hence, the net increase is at most zero; *i.e.*, if arrivals occur to non-zero queues, $F(t)$ can not increase.

- ***Case 2*:** Now we deal with the case when one or more counters at time t were zero. For simplicity, assume all b counters that are incremented are initially empty. For these empty counters, their contribution to $F(t)$ was zero, and their contribution to $F(t+b)$ is db. Again, the counter with the largest count among all N counters is served at time $t+b$. If $F(t) \leqslant bN - db$, then the inductive claim holds trivially. If not, that is, $F(t) > bN - db$, then at least one of the $N-b$ counters that did not get incremented has count i^*+1, such that, $d^{i^*} \geqslant b$; otherwise, it contradicts the assumption $F(t) > bN - db$. Hence, a counter with count at least i^*+1 is served, which decreases $F(t+b)$ by $d^{i^*+1} = db$. Hence the net increase is zero. One can similarly argue the case when arrivals occur to fewer than b empty counters.

Thus we have shown that, for all time t when the counters are served, $F(t) \leqslant bN$. This means that the counter value cannot be larger than i_m, where, $d^{i_m} = Nb$, *i.e.*,

$$C(.) \leqslant \frac{\ln bN}{\ln d}. \tag{9.4}$$

Substituting for d, we get that the counter value is bounded by S. ❐

Theorem 9.4. *(Sufficiency) Under the LCF policy, the number of bits that are sufficient per counter to ensure that no counter overflows, is given by*

$$\boxed{\log_2 \frac{\ln bN}{\ln d}.} \tag{9.5}$$

Proof. We know that in order to store a value x we need at most $\log_2 x$ bits. Hence the proof follows from Theorem 9.3.[6] ❐

[6]It is interesting to note that the cache size is independent of the width of the counters being maintained.

✂Box 9.1: Comparing Buffers and Counters✂

A number can be represented in many notations, *e.g.*, the number 9 is shown in five different notations in Figure 9.3. Most computing systems (servers, routers, *etc.*) store numbers in binary or hexadecimal format. In contrast, humans represent and manipulate numbers in decimal (base-10) format. Each notation has its advantages and is generally used for a specific purpose or from deference to tradition. For example, the cluster-5 notation (a variant of the unary notation) is in vogue for casual counting and tallying, because there is never a need to *modify* an existing value — increments are simply registered by concatenating symbols to the existing count.

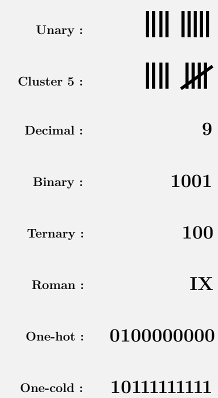

Figure 9.3: *Numeric Notations*

✎**Example 9.4.** Routers sometimes maintain counts in one-hot (or its inverse, one-cold) representation, where each consecutive bit position denotes the next number, as shown in Figure 9.3. With this representation, incrementing a counter is easy and can be done at high speeds in hardware — addition involves unsetting a bit and setting the next bit position, *i.e.*, there are never more than two operations per increment.

Why are we concerned with counter notations? For a moment, assume that counts are maintained in unary — the number two would be denoted with two bits, the number three would have three bits, and so on. This means that each successive counter update simply adds or concatenates a bit to the *tail* of the bits currently representing the counter. Thus, in this unary representation, updates to the counter cache are "1-bit packets" that join the tail of a cached counter (which is simply a *queue* of "1-bit packets").

Viewed in this notation, we can re-use the bounds that we derived on the queue size for the head cache in Chapter 7, Theorem 7.3! We can bound the maximum value of the counters in cache to $S \equiv [Nb(3 + \ln N)]$. Of course, when applying these results, instead of servicing the most deficited queue (as in Theorem 7.3), we would service the longest counter. If we represent these counters in base two, we can derive the following trivially:[a]

Corollary 9.1. *(Sufficiency) A counter of size* $S \equiv \log_2[Nb(3 + \ln N)]$ *bits is sufficient for LCF to guarantee that no counter overflows in the head cache.*

[a] Of course this bound is weaker than Theorem 9.4, because it does not take advantage of the fact that when a counter is flushed it is cleared completely and reset to zero.

9.5 Practical Considerations

There are three constraints to consider while choosing b:

1. Lower bound derived from DRAM access time. The access time of the DRAM is $A_t = Pb/2R$. Hence, if the DRAM supports a random access time T_{RC}, we require $Pb/2R \geqslant T_{RC}$. Hence $b \geqslant 2RT_{RC}/P$, which gives a lower bound on b.
2. Lower bound derived from memory I/O bandwidth. Let the I/O bandwidth of the DRAM be D. Every counter update operation is a read-modify-write, which takes $2M$ bits of bandwidth per update. Hence, $2RM/Pb \leqslant D$, or $b \geqslant 2RM/PD$. This gives a second lower bound on b.
3. Upper bound derived from counter size for LCF policy. From Theorem 9.4, the size of the counters in SRAM is bounded by $\log_2 S$. However, since our goal is to keep only a small-sized counter in the SRAM we need that $\log_2 S < M$. This gives us an upper bound on b.

The system designer can choose any value of b that satisfies these three bounds. Note that for very large values of N and small values of M, there may be no suitable value of b. In such a case, the system designer is forced to store all the counters in SRAM.

9.5.1 An Example of a Counter Design for a 100 Gb/s Line Card

We consider an $R = 100$ Gb/s line card that maintains a million counters. Assume that the maximum size of a counter update is $P = 64$ bytes and that each arriving packet updates one counter.[7] Suppose that the fastest available DRAM has an access time of $T_{RC} = 51.2$ ns. Since we require $Pb/2R \geqslant T_{RC}$, this means that $b \geqslant 20$. Given present DRAM technology, this is sufficient to meet the lower bound obtained on b using the memory I/O bandwidth constraint. Hence the lower bound on b is simply $b \geqslant 20$. We will now consider the upper bound on b.

[7] This could also be an OC192 card with, say, $C = 10$ updates per packet.

✂Box 9.2: Application and Benefits of Caching✂

We have applied the counter caching hierarchy described in this chapter to a number of applications that required measurements on high-speed Enterprise routers at Cisco Systems [33]. At the time of writing, we have developed statistics counter caches for security, logical interface statistics for route tables, and forwarding adjacency statistics. Our implementations range from 167M to 1B updates/s [202] and support both incremental updates ("packet counters") and variable-size updates ("byte counters"). Where required, we have modified the main caching algorithm presented in this chapter to make it more implementable. Also, in applications, where it was acceptable we have traded off reductions in cache size for a small cache overflow probability.

In addition to scaling router line card performance and making line card counter memory performance *robust* against adversaries (similar to the discussion in Box 7.1), in the course of deployment a number of advantages of counter caching have become apparent:

1. **Reduces Memory Cost:** In our implementations we use either embedded [26] or external DRAM [3] instead of costly QDR-SRAMs [2] to store counters. In most systems, memory cost (which is roughly 25% to 33% of system cost) is reduced by 50%.
2. **Wider Counters:** DRAM has much greater capacity than SRAM; so it is feasible to store wider counters, even counters that never overflow for the life of a product.[a]
3. **Decreases Bandwidth Requirements:** The counter caching hierarchy only makes an external memory access once in b time slots, rather than every time slot. In the applications that we cater to, b is typically between 4 and 10. This means that the external memory bandwidth is reduced by 4 to 10 times, and also aids in I/O pin reductions for packet processing ASICs.
4. **Decreases Worst-Case Power:** As a consequence of bandwidth reduction, the worst-case I/O power is also decreased by $4 - 10$ times.

[a]In fact, with ample DRAM capacity, counters can be made 144 bits wide. These counters would never overflow for the known lifetime of the universe for the update rates seen on typical Enterprise routers!

✎**Example 9.5.** We use two different examples for the counter size M required in the system.

1. If $M = 64$, then $\log_2 S < M$ and we design the counter architecture with $b = 20$. We get that the minimum size of the counters in SRAM required for the LCF policy is 9 bits, and this results in

an SRAM of size 9 Mb. The required access rate can be supported by keeping the SRAM memory on-chip.

2. If we require $M = 8$, then we can see that $\forall b, b \geqslant 20, \log_2 S > M$. Thus there is no optimal value of b and all the counters are always stored in SRAM without any DRAM.

9.6 Subsequent Work

The LCF algorithm presented in this chapter is optimal, but is hard to implement in hardware when the total number of counters is large. Subsequent to our work, there have been a number of different approaches that use the same caching hierarchy, but attempt to reduce the cache size and make the counter management algorithm easier to implement. Ramabhadran [203] et al. described a simpler, and almost optimal CMA algorithm that is much easier to implement in hardware, and is independent of the total number of counters, but keeps 1 bit of cache state per counter. Zhao et al. [204] describe a randomized approach (which has a small counter overflow probability) that further significantly reduces the number of cache bits required. Their CMA also keeps no additional state about the counters in the cache. Lu [205] et al. present a novel approach that compresses the values of all counters (in a data structure called "counter braids") and minimizes the total size of the cache (close to its entropy); however, this is achieved at the cost of trading off counter retrieval time, *i.e.*, the time it takes to read the value of a counter once it is updated. The technique also trades off cache size with a small probability of error in the value of the counter measured. Independent of the above work, we are also aware of other approaches in industry that reduce cache size or decrease the complexity of implementation [206, 207, 202].

9.7 Conclusions

Routers maintain counters for gathering statistics on various events. The general caching hierarchy presented in this chapter can be used to build a high-bandwidth statistics counter for any arrival traffic pattern. An algorithm, called largest counter first (LCF), was introduced for doing this, and was shown to be optimal in the sense that it only requires a small, optimally sized SRAM, running at line rate, that temporarily stores the counters, and a DRAM running at slower than the line rate to store complete counters. For example, a statistics update arrival rate of 100 Gb/s on 1 million counters can be supported with currently available DRAMs (having a random access time of 51.2 ns) and 9 Mb of SRAM.

Since our work (as described in Section 9.6), there have been a number of approaches, all using the same counter hierarchy, that have attempted to both reduce cache size and decrease the complexity of LCF. Unlike LCF, the key to scalability of all subsequent techniques is to make the counter management algorithm independent of the total number of counters. While there are systems for which the caching hierarchy cannot be applied (*e.g.*, systems that cannot fit the cache on chip), we have shown via implementation in fast silicon (in multiple products from Cisco Systems [202]) that the caching hierarchy is practical to build. At the time of writing, we have demonstrated one of the industry's fastest implementations [33] of measurement infrastructure, which can support upwards of 1 billion updates/sec. We estimate that more than 0.9 M instances of measurement counter caches (on over three unique product instances[8]) will be made available annually, as Cisco Systems proliferates its next generation of high-speed Ethernet switches and Enterprise routers.

Summary

1. Packet switches (e.g., IP routers, ATM switches, and Ethernet switches) maintain statistics for a variety of reasons: performance monitoring, network management, security, network tracing, and traffic engineering.
2. The statistics are usually collected by counters that might, for example, count the number of arrivals of a specific type of packet, or count particular events, such as when a packet is dropped.
3. The arrival of a packet may lead to several different statistics counters being updated. The number of statistics counters and the rate at which they are updated is often limited by memory technology. A small number of counters may be held in on-chip registers or in

[8]The counter measurement products are made available as separate "daughter cards" and are made to be plugged into a family of Cisco Ethernet switches and Enterprise routers.

(on- or off-chip) SRAM. Often, the number of counters is very large, and hence they need to be stored in off-chip DRAM.

4. However, the large random access times of DRAMs make it difficult to support high-speed measurements. The time taken to read, update, and write a single counter would be too large, and worse still, multiple counters may need to be updated for each arriving packet.
5. We consider a caching hierarchy for storing and updating statistics counters. Smaller-sized counters are maintained in fast (potentially on-chip) SRAM, while a large, slower DRAM maintains the full-sized counters. The problem is to ensure that the counter values are always correctly maintained at line rate.
6. We describe and analyze an optimal counter management algorithm (LCF-CMA) that minimizes the size of the SRAM required, while ensuring correct line rate operation of a large number of counters.
7. The main result of this chapter is that under the LCF counter management algorithm, the counter cache size (to ensure that no counter ever overflows) is given by $\log_2 \frac{\ln bN}{\ln d}$ bits, where N is the number of counters, b is the ratio of DRAM to SRAM access time, and $d = b/(b-1)$ (Theorem 9.4).
8. The counter caching techniques are resistant to adversarial measurement patterns that can be created by hackers or viruses; and performance can never be compromised, either now or, provably, ever in future.
9. We have modified the above counter caching technique as necessary (*i.e.*, when the number of counters is large) in order to make it more practical.
10. At the time of writing, we have implemented the caching hierarchy in fast silicon, and support upwards of 1 billion counter updates/sec in Ethernet switches and Enterprise routers.