Efficient Implementation of a Statistics Counter Architecture

Sriram Ramabhadran
Department of Computer Science & Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
sriram@cs.ucsd.edu

George Varghese
Department of Computer Science & Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
varghese@cs.ucsd.edu

ABSTRACT

Internet routers and switches need to maintain millions of (e.g., per prefix) counters at up to OC-768 speeds that are essential for traffic engineering. Unfortunately, the speed requirements require the use of large amounts of expensive SRAM memory. Shah et al [1] introduced a cheaper statistics counter architecture that uses a much smaller amount of SRAM by using the SRAM as a cache together with a (cheap) backing DRAM that stores the complete counters. Counters in SRAM are periodically updated to the DRAM before they overflow under the control of a counter management algorithm. Shah et al [1] also devised a counter management algorithm called LCF that they prove uses an optimal amount of SRAM. Unfortunately, it is difficult to implement LCF at high speeds because it requires sorting to evict the largest counter in the SRAM. This paper removes this bottleneck in [1] by proposing a counter management algorithm called $LR(T)$ (Largest Recent with threshold T) that avoids sorting by only keeping a bitmap that tracks counters that are larger than threshold T. This allows $LR(T)$ to be practically realizable using only at most 2 bits extra per counter and a simple pipelined data structure. Despite this, we show through a formal analysis, that for a particular value of the threshold T, the $LR(T)$ requires an optimal amount of SRAM, matching LCF. Further, we also describe an implementation, based on a novel data structure called aggregated bitmap, that allows the $LR(T)$ algorithm to be realized at line rates.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.2.3 [**Computer Communication Networks**]: Network Operations—*Network monitoring*; C.2.6 [**Computer Communication Networks**]: Inter-networking—*Routers*

General Terms

Algorithms, Design, Measurement, Performance

Keywords

statistics counter, router

SIGMETRICS'03, June 10–14, 2003, San Diego, California, USA.

1. INTRODUCTION

Packet counting provides a powerful measurement tool [11] for characterizing traffic on service provider networks. Packet counters can be used to perform capacity planning and identify bottlenecks in the network core, to determine the types of packets transiting or destined to the core and the relative ratio of one packet type to another (for example, mail versus FTP), and to analyze attacks by counting packets for commonly used attacks (for example, ICMP request response packets used in a smurf attack). Packet counters can also be used to decide peering relationships. If an ISP A is currently sending packets to ISP C via ISP B and is considering directly connecting (peering) with B, a rational way to decide is to count the traffic destined to prefixes corresponding to B. [3] analyzes the importance of traffic measurement in managing large service provider networks, and motivates the need for more fine-grained measurement capability inside the network.

Legacy routers tend to provide only per-interface counters that can be read by SNMP. Such counters only count the aggregate of all counters going on an interface and make it difficult to estimate traffic AS-AS matrices that are needed for traffic engineering. They can also only be used for crude forms of accounting as opposed to more sophisticated forms of accounting [12] that can count by traffic type (e.g., real time should be more expensive) and destination (some destinations may be routed through a more expensive upstream provider). Thus Juniper networks has introduced filter based accounting [12] where customers can count traffic that matches a rule specifying a predicate on packet header values. Similarly, Cisco provides Netflow based accounting where each 5-tuple can be counted, as well as Express Forwarding commands which allow per-prefix counters [10]. However, enabling these sophisticated accounting and traffic engineering applications, that depend on fine-grained measurement capability, is contingent upon the ability to maintain a large number of statistics counters in Internet routers.

1.1 Why counting is hard

Per-interface counters can be easily implemented because there are only a few counters per interface that can be stored in chip registers. However, doing filter-based or per-prefix counters is more challenging for the following reasons.

- **Large numbers of counters :** Given that even current routers [13] support 500,000 prefixes, and future routers may support more than a million prefixes, millions of real-time counters are potentially required.
- **Multiple counter updates per packet :** A single packet may result in more than one counter being updated such as a flow counter as well as a prefix counter.
- **High speeds :** Line rates have been increasing from OC-192 (10 Gbps) to OC-768 (40 Gbps). Thus each counter matched by a packet must be read and written in the time taken to receive a packet at line speeds. For example, a 40 byte packet must be processed in 8 nsec at OC-768 speeds.
- **Large counter widths :** As line speeds get higher, even 32 bit counters can overflow quickly. To prevent the overhead of frequently polling the router, most vendors [12] now provide 64 bit counters.

A simple inspection of the memory size (1 million counters of 64 bits each is 64 Mb) and bandwidth needs (2 counters of 64 bits each every 8 nsec requires 16 Gbps of memory bandwidth) shows that the task of maintaining a large number of counters at wire speeds is as challenging as other packet processing tasks such as lookup, classification and scheduling. This paper focusses on the task of efficiently maintaining counters in packet switches at line rates.

A model for counter processing is as follows. When a packet arrives at a router line card, some forwarding processor does a lookup on the packet header to determine how to process the packet. The lookup could involve only longest prefix matching to determine the next hop, or a more complex classification step [8] for filter based counters. Suppose that the packet matches prefix i and classification rules j and k. Then the intent is that counters i, j, and k be updated. The counters may be stored in on-chip or off-chip memory and may be updated by the forwarding processor or a separate counter management ASIC. In the latter case, the forwarding processor passes the indices of the counters to be updated to the counter management chip.[1]

The number of counters and the rate at which they can be updated is determined, to a large extent, by the limitations of memory technology. To maintain a large number of counters in SRAM is either infeasible or very expensive. A statistics counter architecture with a million 64 bit counters requires 64 Mb of SRAM. Since SRAM costs are significantly more than DRAM costs (at least a 4:1 ratio), large SRAM requirements increase the costs of line cards considerably. Large SRAM requirements also preclude an on-chip implementation as power and layout considerations limit the amount of SRAM that can be put on-chip. Since SRAM densities are lower than DRAM densities, large SRAM requirements hinder a compact layout, even in an off-chip implementation. Commercial products , that implement counting functionality, are limited by the amount of SRAM they can put on-chip. This limitation is manifested in the form of either lower number of counters or counters of smaller width. For example, the iFlow Accountant product [9] provides either 500,000 counters of width 42 bits or a million counters of width 21 bits. Using on-chip SRAM alone, it is difficult to scale to over a million counters of width 64 bits.

To maintain these counters in off-chip DRAM however, makes it difficult to support counter updates at high line rates. Given the large access times of DRAM, the time to read, update and write a counter would be too large, particularly if multiple counters need to be updated for every packet. A counter update operation involves a memory read followed by a memory write. If the link rate is R Gbps and the minimum packet size is P bits, then memory needs to be accessed every $\frac{P}{2R}$ nsec. For example, in the case of 40 byte TCP packets on a 10 Gbps OC-192 link, the access time is 16 nsec. If every packet causes C counters to be updated, then the effective line rate becomes CR, and memory needs to be accessed every $\frac{P}{2CR}$ nsec. In the same example, if every packet causes two counters to be updated, the access time reduces to 8 nsec. These times are much lower than the access times of commercially available DRAMs (tens of nsec), making it difficult to support counter updates at lines rates by storing the counters in off-chip DRAM.

In essence, balancing the high costs and low densities of SRAM against the large access times of DRAM is an important and non-trivial consideration in the design of a statistics counter architecture.

1.2 Previous Work

In a seminal paper, Shah et al [1] propose a hybrid architecture in which DRAM is used to store the statistics counters but a small amount of SRAM is used to enable counter updates at line rate. If N counters of size M bits are to be maintained, then N counters of full size M bits are stored in DRAM. In addition, N counters of a smaller size $m < M$ bits are cached in SRAM.

The counters in SRAM are used to keep track of recent updates to the counters, and are periodically updated to the corresponding DRAM counters under the control of a *counter management algorithm* (abbreviated CMA). By maintaining counters of smaller size in SRAM, the amount of SRAM required is reduced. By updating the DRAM counters relatively infrequently, the longer access times of DRAM can be tolerated. Figure 1 shows a schematic diagram of the architecture.

While this counter management architecture may superficially resemble conventional caches in processors, the common thread is really only the fact that they both use a faster memory to speed up access to a slower memory. On the other hand, there are the following significant differences, that define a completely different set of metrics and issues than conventional caching.

- **Worst case versus expected case** Processor caches exploit locality of access to optimize the expected case. On the other hand, high line rates mandate counter management to provide small worst case bounds on the time to process a packet.
- **"All of some" versus "some of all"** Processor caches store only a small subset of frequently used data in the cache (all of some). On the other hand, the

[1]Note that by storing the counters at the same relative location as the prefixes or rules, no additional memory is required in the forwarding or classification databases.

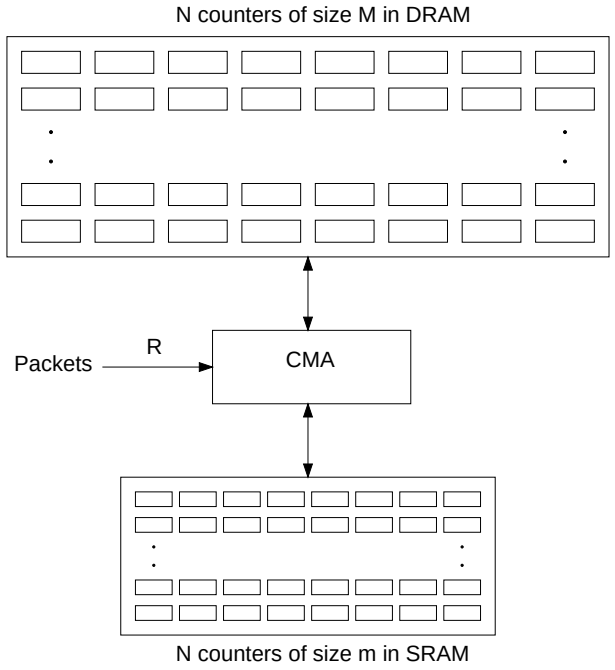

Figure 1: Statistics counter architecture : N counters of size M are maintained in DRAM and N counters of size $m < M$ are maintained in SRAM. The SRAM counters hold recent updates to the counters and are periodically transferred to the corresponding DRAM counters under the control of a counter management algorithm.

counter management architecture stores smaller versions of all the counters in the cache (some of all).

- **Reactive versus proactive replacement** Cache replacement in processors typically occurs only in response to a cache miss. On the other hand, the counter management architecture updates the DRAM counters in parallel to the updates to SRAM. In some sense, counter management is more of a scheduling problem in which a SRAM counter must be updated to the corresponding DRAM counter before it overflows.

Both the amount of SRAM required and the access rate of the DRAM depend on the CMA. The CMA is responsible for picking the order in which SRAM counters are updated to DRAM, and in doing so, must ensure that, no counter overflows. Thus, irrespective of the pattern of traffic arrival, every SRAM counter must be eventually updated to DRAM by the CMA before it overflows. This is not trivial, considering that the SRAM counters are updated at line rates, while the rate at which the DRAM counters can be updated is much smaller. Therefore, for a given CMA and DRAM access rate, the size of the SRAM counter required depends on the maximum value a counter can take under that CMA. If the value a counter can take under a CMA is large, the size of the SRAM counter to prevent overflow is also large and therefore the amount of SRAM required is also large.

Hence an important consideration for a CMA is provably good bounds on the maximum value a counter can take under that CMA. Another equally critical factor is that there should be minimal complexity involved in implementing the CMA. Since the control information required by a CMA needs to be maintained in SRAM, a practical CMA should have minimal storage requirements and also should not require unnecessarily complex hardware.

Shah et al [1] propose a CMA called LCF (Largest Counter First). LCF picks the counter with the largest value to be updated to DRAM. Intuitively, this strategy is optimal because it always updates to DRAM the counter that is closest to overflowing. In fact, LCF is provably optimal in terms of the amount of SRAM required. However due to the necessity of finding the largest among N counter values, it is difficult to implement LCF at high speeds. To quote [1]:

> *But LCF CMA is a complex algorithm to implement at a very high speed. It will be interesting to obtain a similar performance as LCF CMA with a less complex algorithm.*

This paper answers this question [2] by proposing a CMA called $LR(T)$ that is optimal, and therefore has the same performance as LCF. Furthermore, by outlining a relatively inexpensive and simple implementation, it also demonstrates that $LR(T)$ is practical.

1.3 Complexity of Sorting

Why is LCF hard to implement? Clearly, a major bottleneck is finding the highest counter to evict. This is analogous to the requirement in Fair Queuing algorithms such as WFQ [4] to find the packet with the earliest deadline to transmit. Just as round robin and weighted round-robin schemes [5] are more commonly implemented at high speeds, it appears that LCF will have similar implementation issues.

[2]In unpublished research, Shah and Prabhakar [2] have concurrently proposed a different scheme that addresses the same question.

More precisely, one obvious scheme that requires no additional space or hardware complexity is to examine each of the N values. A more efficient scheme would presumably maintain some kind of index data structure that maintains some ordering on the counter values. For example, Bhagwan and Lin [7] describe an implementation of a pipelined heap structure that can determine the largest value at a fairly high expense in hardware complexity and space.

Their P-heap [7] structure is somewhat difficult to pipeline because of the need for parent nodes to read children when processing an update. More importantly, their structure appears to require pointers of size $\log_2 N$ for each counter in the heap just to identify the counter to be evicted. Unfortunately, $\log_2 N$ additional bits per counter can be large (20 for $N = 1$ million) and can defeat the purpose of the CMA, which was to reduce the required SRAM bits from 64 to say 10.

The need for a pointer per heap value seems hard to avoid. This is because the counters must be in a fixed place to be updated when packets arrive, but values in a heap must keep moving to maintain the heap property. On the other hand, when the largest value arrives at the top of the heap, one has to correlate it to the counter index in order to reset the appropriate counter and to banish its contents to DRAM. Notice also that all values in the heap, including pointers and values, must be in SRAM for speed.

By contrast, the counter management architecture we propose in this paper, $LR(T)$ CMA, can be implemented with only 2 bits of additional space per counter and very minimal hardware complexity. This should be contrasted to potentially $\log_2 N$ bits per counter for LCF and increased hardware complexity for maintaining a heap.

Note that reducing space complexity is paramount. Even adding a few extra bits of SRAM per counter could necessitate an implementation with off-chip SRAM. Such a chip would then require pins not only to access the DRAM counters, but to access the SRAM counters as well, potentially creating issues of pin count, power, and board area. On the other hand the 2 extra bits of control overhead plus the 10 or so bits (see numbers later) required per counter, only amounts to 12 Mbits for a million counters which is easily feasible on-chip today.

1.4 Paper Contributions

Our paper proposes a new CMA called $LR(T)$ (Largest Recent with threshold T) that is both optimal as well as easily implementable. For a particular value of the threshold T, $LR(T)$ is analytically shown to require an optimal amount of SRAM to store the counters. Furthermore, $LR(T)$ can be implemented with minimal hardware complexity and only 2 bits of additional space per counter, using a novel data structure called an *aggregated bitmap*.

In essence, $LR(T)$ finesses the need for finding the largest counter by only keeping track of a list of counters above the threshold. $LR(T)$ can be viewed as an approximate bin sorting algorithm that puts counters into two bins (below and above the threshold T), as opposed to LCF which is exact sorting. The interesting property is that, for the purposes of counter management, rough sorting is as good as exact sorting.

To reduce memory, we suggest implementing the list using a bitmap and use a tree data structure to efficiently find bits set to 1. $LR(T)$ requires only slightly more than two SRAM accesses per counter update, thereby making it possible to support multiple counter updates at even OC-768 line rates.

Thus the main contribution of this paper are:

- A new Counter Management Algorithm together with a proof of optimality.
- A demonstration that the SRAM-cache architecture proposed in [1] can be deployed in real routers using an inexpensive and simple implementation. $LR(T)$ can be implemented in an ASIC or integrated into a forwarding processor.
- Solutions to some other problems that need to be solved by a practical counter management algorithm (such as increments greater than 1) that are not addressed in [1].
- A aggregated bitmap data structure that efficiently implements general set membership operations in *pipelined* fashion unlike, say, the structure proposed in [6].

While this paper deals with counter management in the specific context of high-speed networks, maintaining counters is a fundamental problem of measurement. Section 4 briefly discusses the broader applicability of a counter management architecture. The rest of this paper is organized as follows. Section 2 describes and analyzes the $LR(T)$ CMA. Section 3 deals with implementation issues. In particular, it describes the aggregated bitmap data structure and shows how it can be used to implement the $LR(T)$ CMA. Section 4 states our conclusions and proposes directions for future research.

2. $LR(T)$ ALGORITHM

This section describes and analyzes a counter management algorithm called $LR(T)$ (Largest Recent with threshold T). All updates to counters are initially made to the SRAM counters. Every b updates to the SRAM counters, the CMA picks one counter in SRAM to be updated to the corresponding counter in DRAM. The SRAM counter that was updated is then reset to zero. It is assumed that one update to an SRAM counter is made per unit time. At times $t = bk$, an SRAM counter is updated to DRAM and reset. The choice of parameter b is governed by the relative access times of DRAM and SRAM, and is described in detail in section 3.

Algorithm description Let j^* be the counter with the largest value among the counters incremented in the last cycle of b updates to SRAM. Ties may be broken arbitrarily. If the value of this counter $c_{j^*} \geq T$, $LR(T)$ updates counter j^* to DRAM. If $c_{j^*} < T$, $LR(T)$ updates any counter with value at least T to DRAM. If no such counter exists, $LR(T)$ updates counter j^* to DRAM.

$LR(T)$ is actually a family of similar algorithms parameterized by the threshold T. Section 3 shows how to implement $LR(T)$ for any arbitrary threshold T. However two specific values of the threshold $T = 0$ and $T = b$ are of particular interest. A threshold of $T = 0$ allows an extremely simple implementation, while a threshold of $T = b$ is optimal and minimizes the size of SRAM required.

2.1 $LR(0)$ algorithm

Since every counter trivially has a value of at least 0, $LR(0)$ updates to DRAM the counter j^* with the largest value among the counters incremented in the last cycle of b updates to SRAM. $LR(0)$ is a memoryless algorithm in the sense that it remembers only the last b updates to SRAM in determining which counter to update to DRAM. This is in contrast to the $LR(T)$ algorithm with a non-zero threshold T that remembers counters that have exceeded the threshold T.

This memoryless property of $LR(0)$ admits an extremely simple implementation. To implement $LR(0)$, it is sufficient to keep track of the counter with the largest value in the current cycle of updates to SRAM. This can be done easily using an on-chip register that keeps track of the largest counter in the current cycle. An interesting observation is that $LR(0)$ is exactly the same algorithm as $LR(\infty)$ as no counter can reach a value of ∞.

Let C^0_{max} be the maximum value a counter can reach under the $LR(0)$ counter management algorithm. The following theorem provides a lower bound for C^0_{max}.

Theorem 1 : $C^0_{max} \geq \frac{b}{2}(N+1)$

Proof See appendix . □

Theorem 1 implies that a SRAM counter of size at least $\lceil \log_2 (\frac{b(N+1)}{2} \rceil$ is required. For $b = 20$ (a typical figure for the ratio of DRAM and SRAM access rates) and $N = 2^{20}$ (approximately a million), a counter of size at least 24 bits is required. Note that this is only a lower bound on the actual size required. Theorem 1 demonstrates that the extreme simplicity of the $LR(0)$ algorithm comes with the penalty of requiring a relatively large amount of SRAM. While reducing SRAM requirements from 64 bits to 24 bits per counter is still significant, the $LR(b)$ algorithm that we now discuss will do much better.

2.2 $LR(b)$ algorithm

This section provides a theoretical analysis for the $LR(b)$ algorithm in which the threshold has increased from 0 to b, where b is the time between accesses to the DRAM. An upper bound for the largest value a counter can take under the $LR(b)$ algorithm is derived. This determines the amount of SRAM required to implement the algorithm.

Intuition: It the following analysis, it helps to use a potential function argument to bound the value of counters. It is fairly easy to show a bound on a potential function that is equal to the sum of all counters values. Unfortunately, this results in a very poor bound on the value of any one counter. To get a bound that is logarithmic, it helps to use a potential function that is the sum over all counters of an exponential function of the counter value c, such as d^c for some constant d.

It turns out that a reasonable value of d is $\frac{b}{b-1}$ because of the fact that every b units of time, the counter management algorithm is guaranteed to zero out at least 1 counter. Thus as kb time passes only a fraction $(\frac{b-1}{b})^k$ of the total number of counters can stay "large".

Define the *potential* F_j [3] of counter j with value c_j to be

$$F_j = \left\{ \begin{array}{ll} c_j & \text{if } c_j < b \\ \\ \frac{b}{d^b}\, d^{c_j} & \text{if } c_j \geq b \end{array} \right\} \quad (1)$$

where $d = \frac{b}{b-1}$. Note that F_j is continuous at b. Define an aggregate potential function $F(t)$ as the sum of potentials of all counters at time t.

$$F(t) = \sum_{j=1}^{j=N} F_j(t)$$

Lemma 1 : At all times $t = bk$, $F(t) \leq (b-1)(N-1)$

Proof (by induction on time t) We can use induction on time t because we have normalized time such that one SRAM counter is updated every time unit. For the base case, at time $t = 0$, $F(t) = 0$.

For the inductive step, assume that $F(t) \leq (b-1)(N-1)$ at time $t = bk$ for some k. It is required to prove that $F(t+b) \leq (b-1)(N-1)$. Let $C_i, t+1 \leq i \leq t+b$, be the value of the counter incremented at time i. After the counter is incremented, its value is $C_i + 1$. Note that the counter values $C_{t+1}, \ldots, C_{t+b}$ do not necessarily correspond to distinct counters as it is possible that some counters are incremented multiple times between times $t+1, \ldots, t+b$. When the value of a counter is incremented from C_i to C_{i+1}, there is an increase in potential. If $C_i \geq b$, the increase in potential is

$$\frac{b}{d^b}(d^{C_i+1} - d^{C_i}) = \frac{b}{d^b}\, d^{C_i+1}(1 - \frac{1}{d}) = \frac{d^{C_i+1}}{d^b}$$

If $C_i < b$, the increase in potential is $(C_i + 1) - C_i = 1$. At time i, the increase in potential is at most $\max\{1, \frac{d^{C_i+1}}{d^b}\}$. The total increase in potential I between times t and $t+b$ therefore satisfies

$$I \leq \sum_{i=t+1}^{i=t+b} \max\{1, \frac{d^{C_i+1}}{d^b}\} \quad (2)$$

At time $t+b$, let C^* be the value of the counter with largest value among all counters incremented at times $t+1 \ldots t+b$. For all $i, t+1 \leq i \leq t+b$, $C^* \geq C_i + 1$. We now consider two cases.

Case 1 : $C^* \geq b$
The counter with value C^* is updated and reset, causing the potential to decrease by $\frac{b}{d^b}\, d^{C^*}$. The decrease in potential D between times t and $t+b$ satisfies

$$D = \frac{b}{d^b}\, d^{C^*} \quad (3)$$

Now

$$\frac{d^{C^*}}{d^b} \geq \frac{d^b}{d^b} = 1$$

and for all $i (t+1 \leq i \leq t+b)$,

$$\frac{d^{C^*}}{d^b} \quad \geq \quad \frac{d^{C_i+1}}{d^b}$$

[3] Note this potential function is different from the one used in [1] and results in a tighter bound

For all $i, t+1 \leq i \leq t+b,$, $\frac{d^{C^*}}{d^b} \geq \max\{1, \frac{d^{C_i+1}}{d^b}\}$. The change in potential from time t to $t+b$ is given by the difference between the potential increase I and potential decrease D given by equations (2) and (3).

$$\begin{aligned} F(t+b) - F(t) &= I - D \\ &\leq \sum_{i=t+1}^{i=t+b} \max\{1, \frac{d^{C_i+1}}{d^b}\} - \frac{b}{d^b} d^{C^*} \\ &= \sum_{i=t+1}^{i=t+b} (\max\{1, \frac{d^{C_i+1}}{d^b}\} - \frac{d^{C^*}}{d^b}) \\ &\leq 0 \end{aligned}$$

Between times t and $t+b$, there is no net increase in potential and therefore $F(t+b) \leq (b-1)(N-1)$.

Case 2 : $C^* < b$
At time $i, t+1 \leq i \leq t+b$, the increase in potential is $(C_i+1) - C_i = 1$. The total increase in potential between times t and $t+b$ is b. If there is a counter with value at least b, it is updated and reset, causing a decrease of potential of at least $\frac{b}{d^b} d^b = b$. Between times t and $t+b$, there is no net increase in potential and therefore $F(t+b) \leq (b-1)(N-1)$. If there is no counter with value at least b, the counter with value C^* is updated and reset. The potential of this counter is 0 and the potential of each of the remaining $N-1$ counters is at most $(b-1)$ and therefore the aggregate potential $F(t+b) \leq (b-1)(N-1)$.

In both cases, $F(t+b) \leq (b-1)(N-1)$, which proves the inductive hypothesis. □

Let C^b_{max} be the largest value a counter can take under the $LR(b)$ counter management algorithm. The following theorem provides an upper bound for C^b_{max}.

Theorem 2 : $C^b_{max} \leq (2b-1) + \log_d (N-1)$ where $d = \frac{b}{b-1}$

Proof At all times $t = bk$ when a counter is updated to DRAM, $F(t) \leq (b-1)(N-1)$. This implies that the potential of any one counter at $t = bk$ is at most $(b-1)(N-1)$. Therefore the value of any counter at $t = bk$ cannot be larger than

$$\begin{aligned} \log_d (\frac{d^b}{b}(b-1)(N-1)) &= \log_d (d^{b-1}(N-1)) \\ &= (b-1) + \log_d (N-1) \end{aligned}$$

The value of a counter can increase by at most b between two updates to DRAM. Therefore $C^b_{max} \leq (2b-1) + \log_d (N-1)$. □

Shah et al [1] show that there exists a traffic arrival pattern such that a counter can reach a value of $(b-1) + \log_d (N-1)$ under *any* counter management algorithm. Theorem 2 shows that the same quantity $(b-1) + \log_d (N-1)$ is an upper bound for the value a counter can reach at all times $t = bk$ under the $LR(b)$ algorithm. Therefore an upper bound on the maximum value a counter can take under the $LR(b)$ algorithm is a lower bound on the maximum value a counter can take under any counter management algorithm, which proves that $LR(b)$ is optimal in terms of the amount of SRAM required. An additional additive term of b is required to account for increases in counter value between two updates to DRAM. However for practical implementations, b is much smaller than $log_d(N-1)$ and therefore does not change the amount of SRAM required. The upper bound provided by theorem 2 is also tighter than the upper bound in [1] for the LCF counter management algorithm. Since LCF is provably better than any counter management algorithm, the bound for $LR(b)$ applies to LCF as well. Theorem 2 implies that a counter of size $\lceil \log_2 (2b - 1 + \log_d (N-1)) \rceil$ is sufficient. For $b = 20$ and $N = 2^{20}$ (approximately a million), a counter of size $m = 9$ bits is sufficient.

This section showed that the $LR(b)$ CMA is optimal in terms of the amount of SRAM required. The following section demonstrates that the $LR(T)$ algorithm can be efficiently implemented with minimal hardware complexity.

3. IMPLEMENTATION

To implement the $LR(T)$ CMA, it is necessary to locate a counter whose value is at least T. In principle, a list of all counters whose value is at least T could be used. However each element of the list would require a pointer of size $\log_2 N$ to identify a counter. In order to minimize the amount of storage required, an aggregated bitmap data structure is used instead.

3.1 Aggregated bitmap

Consider a fixed universe U of N elements labelled $1, 2, \ldots, N$. It is necessary to record which elements of U are contained in a set S and which are not. A bitmap $b_1 b_2 \ldots b_N$ is used for this purpose where bit b_i is set to 1 if element $i \in S$ and is set to 0 otherwise. Suppose the following membership operations on S are to be implemented.

`add(i)`	Adds element i to set S
`delete(i)`	Deletes element i from set S
`test(i)`	Tests whether element i belongs to set S

The above three operation reduce to a rather straightforward modification or lookup on bit b_i. But suppose the following additional operation is to be implemented.

`find`	Returns any element i that belongs to set S

Finding any element i that belongs to S reduces to searching the bitmap for a bit $b_i = 1$. In the worst case, it may be necessary to examine all N bits of the bitmap. For large N, this is clearly not desirable. To improve the complexity of the `find` operation the following tree structure is constructed that aggregates the information contained in the bitmap. Each group of W bits in the bitmap (W is the word size) is aggregated to form a single node, resulting in a total of $\frac{N}{W}$ nodes. Assuming for simplicity that $\frac{N}{W} = 2^{h}$ [4], these $\frac{N}{W}$ nodes form the leaves of a complete binary tree of height $h+1$. Clearly each internal node is the root of a sub-tree that contains some portion of the entire bitmap. Each internal node in the tree contains two fields called *lcount* and *rcount*, which are defined as follows. For a node whose two children are leaf nodes, the *lcount* field is the number of 1s present in its left child, and the *rcount* field is the number of 1s present in its right child. Note that since both children are leaf nodes, they represent an aggregate of W bits

[4] A reasonable assumption given that in practice, N and W are both likely to be powers of 2

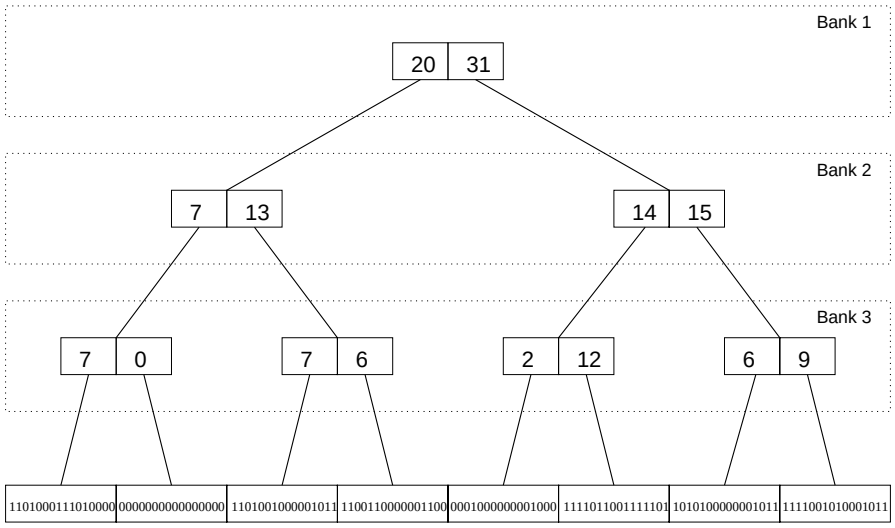

Figure 2: Aggregated bitmap for $N = 128$ elements and $W = 16$ word size. The dotted rectangles indicate that nodes at each level of the tree are stored in different memory banks to facilitate a pipelined implementation

from the bitmap. For a node whose two children are not leaf nodes, the *lcount* field is the sum of the *lcount* and *rcount* fields of its left child, and the *rcount* field is the sum of the *lcount* and *rcount* fields of its right child. A simple inductive argument establishes the invariant that for all internal nodes of the tree, the *lcount* field is the number of 1s present in the portion of the bitmap contained in the sub-tree rooted at its left child, and that the *rcount* field is the number of 1s present in the portion of the bitmap contained in the sub-tree rooted at its right child. Figure 2 shows an aggregated bitmap for $N = 128$ elements and word size $W = 16$.

The set membership operations can now be implemented as a top-down traversal of the tree starting from the root, updating nodes as necessary to maintain the above invariant. Given i, it is trivial to determine whether to follow the left child or the right child at each internal node. During the `add` operation, at each internal node, the *lcount* field is incremented when following the left child and the *rcount* field incremented when following the right child. At the leaf node, bit b_i is set to 1. During the `delete` operation, at each internal node, the appropriate field is decremented while at the leaf node, bit b_i is set to 0. During the `test` operation, no internal node is updated. During the `find` operation, at each internal node, the left child is followed if the *lcount* field is non-zero and the right child is followed if the *rcount* field is non-zero. The invariant guarantees that if one of the fields is non-zero, the corresponding sub-tree has at least one bit $b_i = 1$. If there is no bit $b_i = 1$, both *lcount* and *rcount* fields at the root are zero. The complexity of each operation is proportional to the height of the tree which is equal to $\log_2 \frac{N}{W} + 1$.

Internal nodes in the tree do not explicitly maintain pointers to their left and right children. Rather these are implicitly computed by appropriately laying out the nodes in memory. Therefore each internal node just contains the two fields *lcount* and *rcount* each of which requires $\log_2 \frac{N}{2}$ bits. To ensure that an internal node is contained inside one memory word, and therefore requires only one memory access per read or write, the constraint $W \geq 2 \log_2 \frac{N}{2}$ must be satisfied [5]. Each node in the tree requires W bits of memory and there are $2^{h+1} - 1$ nodes. Therefore the total amount of memory to implement the aggregated bitmap is

$$(2^{h+1} - 1)W = (\frac{2N}{W} - 1)W < 2N$$

or at most 2 bits per element.

[6] proposes a different kind of aggregated bitmap data structure in the context of packet classification. In this scheme, every group of W bits is aggregated into a single bit which is set if any of W bits is set. The aggregation can be repeated to any level, forming a tree. Although this kind of aggregated bitmap can support set operations, it is difficult to pipeline this structure. The reason for this is that the natural flow of `delete` operation is bottom-up and not top-down. In the aggregated bitmap presented in this paper, all operations are top-down, which lends itself to a pipelined implementation.

Pipelined implementation Each of the operations on aggregated bitmap proceeds top-down from one level to another starting at the root. At each level of the tree, there is potentially a memory read followed by a memory write. Storing each of the levels of the tree in a different memory bank permits simultaneous access to all levels of the tree. The

[5] Given the large word sizes possible in hardware nowadays, this is not a serious constraint. For example, for $N = 2^{20}$, even a modest word size of $W = 64$ is sufficient. In fact, large word sizes can be exploited to construct a k-ary tree instead of a binary tree that serves to reduce the height of the tree, improving both the latency and space requirements. In this case, the appropriate constraint is $W \geq k \log_2 \frac{N}{k}$.

relatively small number of levels in the tree (for $N = 2^{20}$ elements and a word size of $W = 64$ the height of the tree is 15) makes this feasible. This allows for a piplelined implementation of the aggregated bitmap operations in which each pipeline stage consists of two memory accesses. This results in an effective throughput of one operation every two memory accesses.

Although this paper proposes the aggregated bitmap data structure in the context of efficiently implementing the $LR(T)$ CMA, it is hoped that the data structure may find uses in other contexts as well. A potential application for the aggregated bitmap is to efficiently maintain the set of active flows in a fair queuing algorithm such as Deficit Round Robin [5].

3.2 Implementation of $LR(T)$

After every b updates to SRAM counters, the CMA selects one SRAM counter to be updated to be updated to the corresponding DRAM counter. To implement the $LR(T)$ CMA, it is necessary to keep track of two things viz. (1) the largest value C^* among all counters updated in the last cycle of b updates along with the corresponding counter j^* and, (2) all counters above the threshold T. The former is accomplished by maintaining an on-chip register in the CMA logic that records the largest value among all counters updated so far in the current cycle. This register may need to be updated every update to SRAM. The latter is accomplished by using an aggregated bitmap with the set S defined as $S = \{j : c_j \geq T\}$ The aggregated bitmap is stored in SRAM distinct from the SRAM used to store the counters. Memory accesses for counter operations and bitmap operations proceed in parallel.

Every update to a SRAM counter requires incrementing that counter in SRAM. In addition, if the value of the counter exceeds the threshold T, the aggregated bitmap data structure must be updated by an `add` operation. Every update to a DRAM counter involves first determining which counter to update, according to the $LR(T)$ algorithm. This is either the counter with the largest value in the last b updates, or any counter with value greater than T. A counter with value greater than T is assumed to be available as a result of a `find` operation initiated at the end of the last DRAM update. In either case, if the value of the counter evicted is greater than T, the bitmap is updated with a `delete` operation. Also a `find` operation is initiated to find a counter with value greater than T that may be evicted after the next b updates. The counter that is chosen is reset and its value updated to the corresponding DRAM counter.

Every cycle of b updates involves b SRAM update operations followed by a single DRAM update operation. An SRAM update operation requires two accesses to update the SRAM counter and potentially two accesses for the `add` operation. A DRAM update operation requires two accesses to read and reset the SRAM counter, and potentially four accesses for the `delete` and `find` operations. By shifting bitmap operations in time, the `delete` operation can be combined with the `find` operation for an integrated complexity of only two memory accesses. A DRAM update operation also requires two DRAM accesses to update the DRAM counter, which can overlap with the next cycle of b SRAM counter updates. Figure 3 shows the sequence of SRAM and DRAM accesses for two successive cycles of b counter updates.

If P is the minimum packet size, R is the link rate and C is the number of counters that are updated per packet, the peak rate of counter updates is $\frac{CR}{P}$. From figure 3, it is clear that every cycle of b updates results in 2 DRAM accesses. Therefore if T_D is the DRAM access time,

$$T_D \leq b\,\frac{P}{2CR} \tag{4}$$

From figure 3, it clear that every cycle of b updates results in $2b + 2$ SRAM accesses. Therefore if T_S is the SRAM access time,

$$\begin{aligned} T_S &\leq \left(\frac{b}{2b+2}\right)\frac{P}{CR} \\ &= \left(\frac{b}{b+1}\right)\frac{P}{2CR} \end{aligned} \tag{5}$$

Constraints 4 and 5 impose lower bounds on the choice of b. Given SRAM with access time T_S and DRAM with access time T_D, it is desirable to choose as small a value of b subject to constraints 4 and 5 to minimize the amount of SRAM required. Fixing the value of b results in an SRAM counter of size $m = \log_2(2b-1) + \log_d(N-1)$ as per Theorem 2. An additional 2 bits of SRAM per counter are required for maintaining the aggregated bitmap. The total SRAM requirements are therefore $(m+2)N$.

3.3 Large counter updates

The analysis in the preceding sections assume that each update causes a counter to be incremented by 1. While this works fine for packet counters, it does not work for byte counters that must be incremented by the number of bytes in each packet. Each counter can then be increased by maximum packet size on each packet arrival.

Suppose the size of the maximum increment per update is u. The analysis is similar with the bound provided by Theorem 2 having an extra multiplicative factor of $\log_2 u$. Thus to support the same DRAM access rate b, the $LR(b)$ CMA requires $\log_2 u$ additional bits per SRAM counter. When u is large, these additional bits can be significant. A counter counting the total bytes due to Ethernet packets would need to support increments of up to 1500 bytes, thereby requiring 11 additional bits.

One way to avoid this overhead is to use a probabilistic scheme to update the counters. More precisely, suppose that when counter j gets updated by amount x, we increment the value of counter j by 1 with a probability of $\frac{x}{u}$. Note that in this scheme, the value of a counter increases by at most one every update.

A sequence of updates $x_1, x_2, \ldots, x_n$ to the same counter results in a counter value $X = \sum_{i=1}^{i=n} X_i$ where each X_i, is a 0-1 random variable with $\Pr[X_i = 1] = \frac{x_i}{u}$. The expected value of the counter after n updates is

$$\mu = E[X] = E[\sum_{i=1}^{i=n} X_i] = \sum_{i=1}^{i=n} E[X_i] = \sum_{i=1}^{i=n} \frac{x_i}{u} = \frac{1}{u}\sum_{i=1}^{i=n} x_i$$

Therefore the value of the counter X multiplied by u gives an approximation of the required value $\sum_{i=1}^{i=n} x_i$, which is accurate in an expected sense. The following Chernoff bounds can be used to derive a high confidence bound on the value

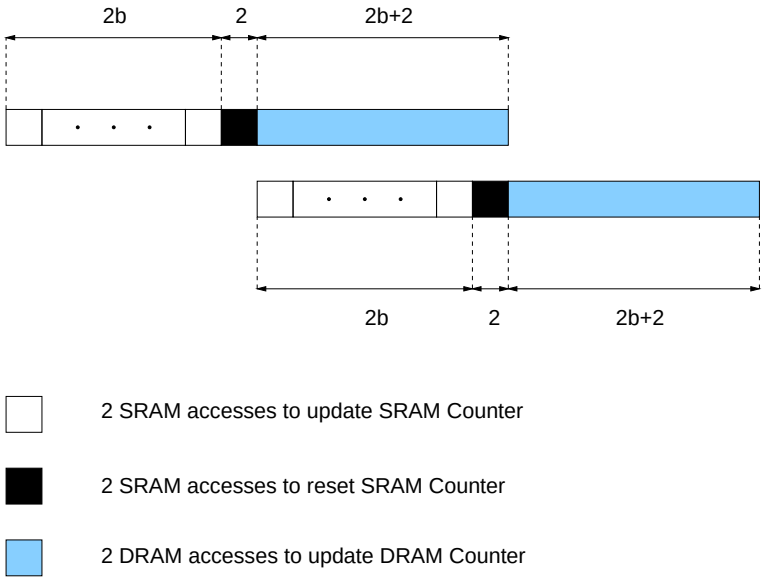

Figure 3: Timing diagram for SRAM and DRAM updates for two successive cycles of b counter updates. Note that SRAM accesses for counter operations and bitmap operations are overlapped

of X. For all $0 < \beta \leq 1$,

$$\Pr[X \leq (1-\beta)\mu] \leq e^{-\frac{\beta^2\mu}{2}}$$
$$\Pr[X \geq (1+\beta)\mu] \leq e^{-\frac{\beta^2\mu}{3}}$$

The probability that the approximation uX lies outside a multiplicative factor of β of the required $\sum_{i=1}^{i=n} x_i$ drops exponentially with the expected value of the counter. Thus when updates accumulate over a period of time, the approximation tends to become more and more accurate.

4. CONCLUSION

Packet switches need to maintain counters for collecting statistics on various events. Shah et al [1] introduce a statistics counter architecture that can support a large number of counters at line rates. The basic idea is to use small width SRAMs as a cache that must be backed up by larger width DRAMs, thus combing the speed of SRAMs with the cheapness and density of DRAMs. [1] also proposes a counter management algorithm called LCF that is shown to require an optimal amount of SRAM by periodically dumping the largest counter in SRAM to DRAM.

However, it is difficult to implement LCF at high speeds because of the need to have logic that selects the largest SRAM counter to evict. Our paper removes this "sorting" bottleneck in [1] by proposing a counter management algorithm called $LR(T)$ (Largest Recent with threshold T) that is not only optimal but also practically realizable. Through a formal analysis, our paper shows that for a particular value of the threshold T, the $LR(T)$ requires an optimal amount of SRAM despite its simpler structure that avoids computing the largest counter. Further, our paper also describes an implementation, based on a novel data structure called aggregated bitmap, that allows the $LR(T)$ algorithm to be realized at line rates.

Table 1 compares the naive approach of implementing all counters in SRAM with the LCF and $LR(b)$ schemes. For a reference system of a million counters operating at 10 Gbps with 10 updates per packet, $LR(b)$ can be implemented using roughly 11 bits of SRAM per counter and simple logic. When compared to potentially 64 bits of SRAM per counter for a naive solution, this is nearly a 5-fold reduction in SRAM size. This will result in nearly a factor of two reduction in overall system cost after counting the extra DRAM added. $LR(b)$ is also much easier to implement than LCF due to the sorting requirement that increases both the implementation complexity as well as the amount of memory required to maintain state.

We see no reason why simple counter management logic cannot be deployed (either as a stand-alone chip with interfaces to external DRAM or within a forwarding processor) using our $LR(b)$ algorithm.Given that SRAM contributes a sizeable fraction of the line card cost and power, practical counter management algorithms could be widely used in the future.

In summary, while [1] took the fundamental step of proposing a general framework for maintaining statistics counters via caching, our paper brings the framework to fruition by demonstrating an implementation of a CMA that can be realized at line rates. We stress the practicality of our scheme since statistics counters are a crucial requirement to enable applications like accounting, security and traffic engineering, that require fine-grained measurement in the network core.

[6]The control memory required for LCF is estimated assuming that any indexing structure that maintains counters in heap order requires at least $\log_2 N$ bits per counter. While it is certainly conceivable that LCF could be implemented with less control memory, it is unlikely that such an implementation could come close to the 2 bits per counter required by the $LR(T)$ implementation described in this paper.

[7]Cost is estimated relative to the naive implementation, based on the amounts of SRAM and DRAM required by each scheme, assuming a 4:1 cost ratio.

	Naive	*LCF*	*LR*(*b*)
Counter memory	64 Mb SRAM	9 Mb SRAM 64 Mb DRAM	9 Mb SRAM 64 Mb DRAM
Control memory	None	20Mb SRAM [6]	2 Mb SRAM
Control logic	Simple	Hardware heap	Aggregated bitmap
Implementation complexity	Low	High	Low
Cost [7]	100 %	70 %	42 %

Table 1: Cost - benefit comparison for different schemes for a reference system with a million 64-bit counters and a line rate of 10 Gbps with 10 counter updates per packet

Future directions Finally, we comment on the broader applicability of the work described in this paper and propose directions for future research. While for the purpose of solving a very concrete real-world problem, this paper focusses on counter management in a very specific setting, viz. statistics gathering in high-speed Internet routers, counters are per se a universally used mechanism in any kind of measurement or statistics collection. Counter management could therefore be an important issue in other application domains as well.

For example, consider a busy web server that maintains statistics on how many times a particular user accesses a particular piece of content. The statistics counters are maintained in main memory for quick access, but need to be archived to permanent disk storage in case of failure. To minimize overhead, the archiving process runs only periodically and selects only a fraction of the large number of statistics counters to be transferred to disk. To minimize information loss in case of failure, counters with large values are transferred to disk in preference to counters with small values. Thus we see that counter management is important in a very different setting with a different memory hierarchy (main memory versus disk) and different tradeoffs (overhead versus information loss). Note that in this particular scenario, an $LR(T)$ - like algorithm could be used to keep track of "large" counters that are above some threshold T. In a software setting, since space is no longer a primary consideration, $LR(T)$ could be implemented in constant time complexity using a list instead of a bitmap.

The above example is meant to illustrate that specialized counter management architectures, such as the one discussed in this paper , could be valuable in other domains as well, with design issues specific to each domain. Keeping this in mind, we conclude by posing questions that could be addressed by future research.

- Are there other counter management algorithms or architectures whose integrated (counter and control) SRAM requirements are lower ? What is the theoretical minimum memory required ?

- What is the effect of multiple levels of memory hierarchy ? How can counter management be efficiently implemented in different memory hierarchies with different tradeoffs ?

- How can locality be used to optimize counter management ? For example, can a Zipf distribution of updates be used to reduce the amount of SRAM required while still providing high confidence worst case bounds on the time required to process an update ?

5. ACKNOWLEDGEMENTS

The work of the second author was supported by a grant from NIST for the Sensilla project and NSF Grant ANI 0137102.

6. REFERENCES

[1] D. Shah, S. Iyer, B. Prabhakar, and N. McKeown. Maintaining statistics counters in router line cards. In *IEEE Micro*, 2002.

[2] D. Shah, Personal communication, October 2002

[3] M. Grossglauser and J. Rexford. Passive traffic measurement for IP operations. To appear as a chapter in *The Internet as a Large-Scale Complex System*, Oxford University Press, 2002

[4] A. Demers and S. Shenker. Analysis and simulation of a fair queuing algorithm. In Proceedings of *ACM SIGCOMM 89*, 1989.

[5] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round robin. In Proceedings of *ACM SIGCOMM 95*, 1995.

[6] F. Baboescu and G. Varghese. Scalable packet classification. In Proceedings of *ACM SIGCOMM01*, 2001.

[7] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In Proceedings of *IEEE INFOCOM 00*, 2000.

[8] P. Gupta and N. McKeown. Algorithms for packet classification. In *IEEE Network*, 2001.

[9] iFlow Accountant. `www.siliconaccess.com/products/`

[10] Cisco Express Forwarding Commands `www.cisco.com/univercd/cc/td/doc/product/software/ios120/12supdoc/12cmdsum/12csswit/cscef/htm.`

[11] Internet processor II asic : Visibility into network operations. `www.juniper.net/techcenter/appnote/350002.html`.

[12] Juniper networks solutions for network accounting - white paper. `www.juniper.net/techcenter/appnote/350003.html`.

[13] Tests prove reliability and scalability of juniper networks ip vpn solutions. `http://biz.yahoo.com/bw/020501/10140 3.html`.

APPENDIX

Theorem 1 : $C^0_{max} \geq \frac{b}{2}(N+1)$

Proof A traffic arrival pattern $\pi_n(i_1, i_2, \ldots, i_n)$ is constructed that updates counters $i_1, i_2, \ldots, i_n$ such after time $t = |\pi_n|$, at least one of the counters has a value of at least $\frac{b}{2}(n-1)$. The required traffic arrival

pattern is constructed inductively. Define

$$\pi_2(i_1, i_2) = \underbrace{i_1, i_1, \ldots, i_1}_{\frac{b}{2}\ times}\ \underbrace{i_2, i_2, \ldots, i_2}_{\frac{b}{2}\ times}$$

After $t = |\pi_2|$, both i_1 and i_2 have a value of at least $\frac{b}{2}$. One of them is updated to DRAM and reset by the $LR(0)$ CMA but the other one still has a value of at least $\frac{b}{2}$.
Consider the traffic arrival pattern

$$\pi_{n-1}(i_1, i_2, \ldots, i_{n-1})$$

By the inductive hypothesis, after $t = |\pi_{n-1}|$, at least one of the counters $i_1, i_2, \ldots, i_{n-1}$ has a value of at least $\frac{b}{2}(n-2)$. Without loss of generality, assume that this counter is i_1. Now consider the traffic arrival pattern

$$\pi_{n-1}(i_2, i_3, \ldots, i_n)$$

By the inductive hypothesis, after $t = 2|\pi_{n-1}|$, at least one of the counters $i_2, i_3, \ldots, i_n$ has a value of at least $\frac{b}{2}(n-2)$. Without loss of generality assume that this counter is i_2. Note that since i_1 was never updated during this period, it still has a value of $\frac{b}{2}(n-2)$. $LR(0)$ never updates counters that were not incremented in the last cycle of b updates. Now consider the traffic arrival pattern

$$\underbrace{i_1, i_1, \ldots, i_1}_{\frac{b}{2}\ times}\ \underbrace{i_2, i_2, \ldots, i_2}_{\frac{b}{2}\ times}$$

After $t = 2|\pi_{n-1}| + b$, both i_1 and i_2 have a value of at least $\frac{b}{2}(n-2) + \frac{b}{2} = \frac{b}{2}(n-1)$. One of them is updated to DRAM and reset by the $LR(0)$ CMA but other one still has a value of at least $\frac{b}{2}(n-1)$. Now consider the traffic arrival pattern

$$\begin{aligned}\pi_n(i_1, i_2, \ldots, i_n) = {} & \pi_{n-1}(i_1, i_2, \ldots, i_{n-1}) \\ & \pi_{n-1}(i_2, i_3, \ldots, i_n) \\ & \underbrace{i_1, i_1, \ldots, i_1}_{\frac{b}{2}\ times}\ \underbrace{i_2, i_2, \ldots, i_2}_{\frac{b}{2}\ times}\end{aligned}$$

After $t = |\pi_n| = 2|\pi_{n-1}| + b$, it has been shown that there is a counter with value at least $\frac{b}{2}(n-1)$, proving the inductive hypothesis. Therefore the arrival pattern $\pi_N(1, 2, \ldots, N)$ results in one counter that has a value of at least $\frac{b}{2}(N-1)$. In the next cycle of b updates, the value of this counter can increase by b, giving a counter value of at least $\frac{b}{2}(N+1)$. □