Tabulation Based 4-Universal Hashing with Applications to Second Moment Estimation

Mikkel Thorup *Yin Zhang*

AT&T Labs—Research, Shannon Laboratory
180 Park Avenue, Florham Park, NJ 07932, USA.
(mthorup,yzhang)@research.att.com

Abstract

We show that 4-universal hashing can be implemented efficiently using tabulated 4-universal hashing for characters, gaining a factor of 5 in speed over the fastest existing methods. We also consider generalization to k-universal hashing, and as a prime application, we consider the approximation of the second moment of a data stream.

1 Introduction.

This paper is about fast 4-universal hashing, with fast data streaming algorithms being the prime application. We also consider generalization to k-universal hashing for arbitrary k. For any $i \geq 1$, let $[i] = \{0, 1, \cdots, i-1\}$. As defined in [19], a class $\mathcal{H}$ of hash functions from $[n]$ into $[m]$ is a *k-universal class of hash functions* if for any distinct $x_0, \cdots, x_{k-1} \in [n]$ and any possibly identical $v_0, \cdots, v_{k-1} \in [m]$,

$$\Pr_{h \in \mathcal{H}} \{h(x_i) = v_i, \ \forall i \in [k]\} = 1/m^k \tag{1.1}$$

By a *k-universal hash function*, we mean a hash function that has been drawn at random from a k-universal class of hash functions. Our main contribution is a fivefold speed-up for 4-universal hashing of keys consisting of one or a few words.

Hashing is typically applied to a set of $s \ll n$ keys from $[n]$ and often we consider collisions harmful. For any $k \geq 2$, with k-universal hashing, the expected number of collisions is bounded by s^2/n collisions. However, with 2-universal hashing, the variance in the number of collisions may be as large as $\Theta(s^4/n)$. On the other hand, as shown in [7], with 4-universal hashing, the variance is no bigger than the expected number of collisions. As described in [7] this means that 4-universal hashing is more reliable in many algorithmic contexts.

Our special interest in fast 4-universal hashing is due to its applications in the processing of data streams, an area that has recently gathered intense interest both in the theory community and in many applied communities such as those of databases and the Internet (see [2, 13] and the references therein). A common data stream scenario is as follows. A large stream of items arrive one by one at a rapid pace. When an item arrives, we have very little time to update some small amount of local information that we maintain on the data stream. This information is selected with a certain question or type of questions in mind. The item itself is lost before the next item arrives. When all the items have passed by, we have to answer questions on the data stream.

A classic example is the second moment computation from [1]. Each item has a weight and a key and we want to compute the second moment which is the sum over each key of the square of the total weight of items with that key. The generic method used in [1] is that when an item arrives, a hash function is computed of the key, and then the hash value is used to update the local information. At any point in time, the local information provides an unbiased estimator of the second moment of the data stream seen so far. In order to control the variance of the estimator, the hash function used in [1] has to be 4-universal.

The generic method from [1] is used in many other streaming algorithms. Sometimes we just use 2-universal hashing. Other times we have a choice between simpler algorithm using 4-universal hashing and a more complicated one using 2-universal hashing. And finally, as in the second moment example, we only have algorithms using 4-universal hashing.

In many of the streaming algorithms, computing the hash function is a bottleneck. The basic reason to prefer 2-universal hashing over 4-universal hashing is that it is an order of magnitude faster with existing methods. However, here we improve the speed of 4-universal hashing by a factor of 5, making it a more viable option in time critical scenarios.

1.1 Concrete application areas. The application [11] that originally motivated us for this research was a sniffer that monitors packets passing through a high speed Internet backbone link at OC48 speed (2.48 gigabits per second). At such high link speed, it is common for packets to arrive at a rate of more than a million per second, thus leaving us with less than a microsecond per packet. In the worst case, when all packets are of the minimum Internet packet size of 40 bytes (320 bits), the packet arrival rate can be as high as $2.48 \times 10^9/320 = 7.75 \times 10^6$ per second, leaving us with less than 130 nanoseconds per packet. A critical part of the application was to compute the second moment with the packet key being its single word 32-bit IP address, and the packet weight being its size in bytes. The speed-up achieved in this paper made the application possible.

We note that independent of computer speeds, there is an absolute virtue in processing information with just a few instructions. The basic point is that many streams are already passing through computer software, hence limited by the actual processor's speed. If we can process data in a few instructions, then chances are that we are as fast as everything else, hence that we can keep up with the stream. An example of such software limited streams are IP firewalls that are often implemented in software. Another example is the flow level statistics exported by most IP routers.

1.2 Tabulation based hashing. On most of today's computers, the fastest way of generating a 2-universal bit string from a key is to divide the key into characters, use an independent tabulated 2-universal function to produce a bit string for each character, and then just return the bit-wise exclusive-or of each of these strings. This method goes back to [4], and an experimental comparison with other methods is found in [16]. More precisely, if $\mathcal{H}$ is a 2-universal class of hash functions from characters to bit-strings, and we pick q independent random functions $h_0, \cdots, h_{q-1} \in \mathcal{H}$, then the function $\vec{h}$ mapping $a_0 a_1 \cdots a_{q-1}$ to $h_0[a_0] \oplus h_1[a_1] \cdots \oplus h_{q-1}[a_{q-1}]$ is 2-universal. Here $\oplus$ denotes bit-wise exclusive-or. If $\mathcal{H}$ is 3-universal, then so is h. However, the scheme breaks down above 3. Regardless of the properties of $\mathcal{H}$, $\vec{h}$ is not 4-universal.

Above we used '[]' around the arguments of the h_i to indicate that the h_i are tabulated so that function values are found by a single look-up in an array.

1.2.1 Our results. Despite the above obstacles, we show in this paper that it is possible to use tabulation for fast 4-universal hashing. As a simple illustration, consider the case where keys are divided into two characters. Then we will show that

$$h[ab] = h_0[a] \oplus h_1[b] \oplus h_2[a+b]$$

is a 4-universal hash function if h_0, h_1, and h_2 are independent 4-universal hash functions into strings of the same length. As a slight caveat of the above scheme is that the derived character $a+b$ requires one more bit than a and b, hence that h_2 need to be over a domain of twice the size. It would have been nicer if we could just apply h_2 to $a \oplus b$ instead of $a+b$, but then we will show that the combined function is not 4-universal. We can reduce the domain of $a+b$ by performing the addition over an appropriate odd prime field. With character length $c = 8, 16$, we exploit that 2^c+1 prime. Then the domain of the derived character $a+b$ is only one bigger than that of the input characters.

The above scheme could be applied recursively, but then, for q characters, we would end up using $q^{\log_2 3}$ hash functions. We show here that we can get down to $2q-1$ hash functions whose output strings need to be $\oplus$ed. Apart from hashing q input characters in $[2^c]$, we hash $q-1$ derived characters over $[2^c+q]$. The derived characters are all generated using a total of $q+4$ simple operations over integers returned for free as part of the look-up done over the input characters.

We also present a scheme for general k that gives k-universal hashing using $(k-1)(q-1)+1$ k-universal hash functions. For $k=4$, this is not as good as the previous result, but it does have the advantage that the derived characters have exactly the same length as the input characters.

As a theoretical comment, we note that our scheme completely avoids multiplication. It only uses $O(kq)$ AC^0 operations like addition, shifts, and bit-wise Boolean operations plus memory look-ups. For contrast, we know that any small space implementation of k-universal hashing needs non-AC^0 operations like multiplication [12]. The space of our tables allows us to circumvent this problem. An alternative solution would have been to use our space to tabulate multiplication of $c/2$-bit characters, and use this table to implement multiplication over ℓ-bit keys with AC^0 operations. Even with the asymptotically fastest multiplication [15], we would use $\omega(q \log q)$ look-ups per multiplication and $\omega(kq \log q)$ look-ups for k-universal hashing. This is worse than the $O(kq)$ look-ups with our solution, and it is not even remotely practical.

1.2.2 Random derived characters. Recently and independently, a somewhat similar scheme has been suggested [9, §5], but where the derived characters are based themselves on random hashing. As we shall see below, our deterministically derived characters work much better.

The scheme in [9] takes an integer parameter p, and generates p independent universal hash functions $f_0, \ldots f_{p-1}$ from input keys to c-bit characters. Here, as in [4], *universal* just means that for any inputs x and y, $\Pr\{f_i(x) = f_i(y)\} = 1/2^c$. For 32- and 64-bit input words, such a universal hash function can be implemented with a multiplication and a shift [8]. The output is defined as

$$h(x) = h_0[f_0(x)] \oplus h_1[f_1(x)] \oplus \cdots \oplus h_{p-1}[f_{p-1}(x)].$$

In [9, Proof of Theorem 5], they show for any distinct input $x_0, \ldots, x_{p-1}$ and output $y_0, \ldots, y_{p-1}$

$$\Pr\{h(x_i) = v_i,\ \forall i \in [k]\} = 1/m^k + (k/(p+1))(k/2^c)^p$$

Above, the additive term $(k/(p+1))(k/2^c)^p$ is the error relative to d-universality as defined in (1.1). In order to get remotely k-universal, we set $2^{cp} = m^k$ and accept an error factor of $k^{p+1}/(p+1)$. For $m = 2^\ell$ and $c = \ell/q$, this means that $p = kq$ where p is the number of hashed characters and table look-ups. This is actually more than the $(k-1)(q-1)+1$ deterministically derived characters that we use for our general perfectly k-universal scheme.

Our special scheme for 4-universal hashing provides even bigger improvements, using only $2q-1$ as opposed to $4q$ derived characters and look-ups, and avoiding the error factor $k^{4q+1}/(4q+1)$. Conversely, if we limit the scheme from [9] to use the same computational resources as we do, that is, only $p = 2q-1$ randomly derived characters and look-ups, the error is

$$(k/(p+1))(k/2^c)^p 2^{4\ell} > (8^{2q}/2q)2^{2\ell} > m^2$$

For example, hashing 64-bit integers to 64-integers using 16-bit characters, [9] gets an error factor $> 2^{149}$. In all fairness, it should be mentioned that the scheme in [9] is not claimed to be practical. Also, the analysis in [9] is geared towards asymptotic bounds, and it may be possible to tighten it.

We note that [9, §5.2] stipulates that they can implement their k-universal hashing with a single multiplication, though over numbers that are q times as long as the input keys. As mentioned previously, our scheme avoids multiplication altogether.

1.3 Hashing single and double words in C. The focus of this paper is to develop efficient C code for 4-universal hashing of single and double words of 32 and 64 bits, respectively, producing a corresponding number of output bits. Indeed, we end up gaining a factor 5 in speed over previous methods. We shall later return to the case where input and output are of different sizes.

Concerning other key lengths, if keys have less than 16 bits, we can hash them trivially using a complete table over all such keys. If keys have between 16 and 32 bits, we hash them as 32 bit keys. If keys have between 32 and 64 bits, we hash them as 64 bit keys. Finally, if keys have more than 64 bits, we first apply fast standard universal hashing into 64 bits, and then we apply 4-universal hashing on the reduced keys.

In the above reduction to 64 bit keys, the universal hashing means that two keys get the same reduced key with probability 2^{-64}. Hence, if there are $n \ll 2^{32}$ different keys in the stream, they will all have distinct reduced keys. However, as detailed in [18, A3], if the target is to estimate the second moment, then the error is small with high probability if most of the mass of the stream is distributed on $n \ll 2^{64}$ keys, which is most certainly the case in any foreseeable future.

We note that our techniques generalize perfectly to 96 and 128 bits if that is of any interest. The point we try to make here is that a more standard asymptotic analysis for keys of non-constant length, e.g., favoring Schönhage-Strassen multiplication of large numbers [15], is much less relevant for the kind of streaming applications we have in mind.

Another point in not considering the asymptotic case is that our worst competitor may simply be undefined, depending on a major unresolved problem in number theory.

1.4 The opposition. When implementing k-universal hashing from words to words in C on a standard computer, our worst competitor is the original function from [19]:

$$h(x) = \sum_{i=0}^{k-1} a_i x^i \bmod p \tag{1.2}$$

for some prime $p > x$ with each a_i picked randomly from $[p]$. If p is an arbitrary prime, this method is fairly slow because the 'mod p' is slow. However, as pointed out in [4], we get a fast implementation if p is a so-called Mersenne prime of the form $2^i - 1$. Then (1.2) gives the fastest known 4-universal hashing on a processor with standard arithmetic operations. We shall refer to this as *CW-trick*. In the hashing of 32-bit integers, we can use $p = 2^{61} - 1$, and for 64-bit integers, we can use $p = 2^{89} - 1$.

As mentioned previously, we do not know if CW-trick is defined for arbitrary key lengths, for it is a major open problem in number theory if arbitrarily large Mersenne primes exist. The largest known so far is $2^{13466917} - 1$.

For the second moment estimation from [1], it is actually preferable that the 4-universal output is a bit string, for then each bit position is a 4-universal bit independent of the remaining string. In contrast, if the output is in a prime field, the bit positions are not independent, and then we typically have to give up some of the most significant bits in order to get an approximately 4-universal bit string. The output of our 4-universal hashing is a bit string as desired. Also, we can easily generate long 4-universal bit-strings. All we have to do is to put long bit-strings in the 4-universal character tables, and then $\oplus$ these bit-strings as we look them up in a sequential read.

We note that a weakness of our tabulation based scheme relative to CW-trick is that we require fairly large pre-computed tables whereas CW-trick just requires access to $a_0, \ldots, a_{k-1}$. One can easily imagine applications where it is desirable to compute hash values directly from a small space representation of a hash function. However, for our streaming applications, it is not a problem to initialize some tables in an up-start face.

We are going to compare CW-trick with our tabulation based method both based on a high level instruction count, and based on experiments on two different computers.

1.4.1 With p a power of 2. In [6] it was shown that (1.2) can be used with p an appropriately large power of two, outputting a suffix of the result as a k-universal bit string. For 2-universal hashing, this power-of-two scheme has proved very fast [16]. However, for 4-universal hashing, the scheme needs $p = 2^{218}$ just to hash from 32 bits to 32 bits [6, Theorem 10]. The multiplication of 218-bit integers makes this method much slower than CW-trick for 4-universal hashing.

We note that since the method from [6] gives us 4-universal bit strings, we could use it to initialize the character tables needed in our tabulation based method. Assuming that the data stream is much bigger than the tables, the initialization time is not an issue.

1.5 The C-level instruction count. Besides an experimental comparison, we are going to compare our tabulation based scheme with CW-trick using a coarse-grained analysis of C code. We assume we have a 64-bit processor, and we charge a unit cost for each instruction on one or two 64-bit double word. Of computational instructions we have standard arithmetic operation such as addition and multiplication. We note for both addition and multiplication that overflow beyond the 64 bits are discarded. In particular, this means that multiplication is only used to multiply integers below 2^{32}. With CW-trick we do not need modulus or division so we do not need to worry about the slowness of these operations. Of other computational instructions, we have regular and bit-wise Boolean operations and shifts. Finally, we may define a vector of characters over a double word and extract a character at unit cost.

We note that among the above operations, multiplication is typically the most expensive. Since multiplication is used by CW-trick and not by us, we are generous to the opponent when only charging one unit for multiplication.

We assume that our processor has a small number, say 30, of registers. Here registers are just thought of as memory that is so fast that copying between cells is almost free. Since the registers are controlled by the compiler, it is convenient to ignore it. When running CW-trick, we assume that all variables reside in registers. However, we do charge a unit cost for memory access beyond the registers. This means that our tabulation based methods are charged a unit for each look-up. Obviously, the unit cost is only fair if the tables are small enough to fit in reasonably fast memory. For example, if tables over 16-bit characters were too slow, we could switch to tables over 8-bit characters.

As a final cost, we charge a unit for a jump. For a conditional jump, we charge for the evaluation of the condition and for the jump if it is made. All our procedure calls are inline, so we do not need to charge for them.

We refer to the above cost as the *C-level instruction count*. Here "C-level" refers to the fact that a finer analysis would have to take the concrete machine and compiler into account. Our C-level analysis will be complemented with an experimental evaluation on two different computers, and as it turns out, the C-level instruction count does give a fairly accurate prediction of the actual running times.

1.5.1 Modern processors. For the k-universal hashing of longer keys, it is worth noting that our algorithms are ideally suited for the kind of vector operations supported supported on 128-bit words by modern processors like the Pentium 4 [10, 17]. In fact, we are really coding such vector operations, making sure that we have enough space between coordinates that we do not get overflow from one coordinate to another. For contrast, these modern processors do not support full-word multiplication, and hence they do not help as much for the traditional method in (1.2).

1.6 Second moment estimation. The estimation of the second moment is a canonical application of our tabulation based 4-universal hashing. We present a new estimator that given a 4-universal hash value, yields the same variance and space gains roughly a factor of 2 over the best combination of previous methods [1, 5]. In [11], this speed is used, as part of a larger application, in real time estimation of the second moment over IP-addresses of packets coming through a high speed Internet router. Our second moment estimator is described in §4, deferring some details to [18, §A].

2 Tabulation based hashing.

In this section, we show how tabulation can be used for fast 4-universal hashing. First we present a general framework for k-universal hashing along with some simple lemmas. Next, we present a scheme for 4-universal hashing on two input characters that requires 3 table lookups. We then generalize the scheme to achieve 4-universal hashing on q input characters with $2q-1$ table lookups. We also present a scheme for general k that gives k-universal hashing on q characters using $(k-1)(q-1)+1$ table lookups. We then compare our 4-universal hashing scheme with *CW-trick*, the fastest known algorithm, both in terms of C-level instruction count and actual running times. The results suggest that our tabulation based scheme consistently wins by at least a factor of 5.

2.1 General framework. Our general framework for tabulation based k-universal hashing with q characters is as follows.

1. Given a vector of q *input characters* $\vec{x} = (x_0\ x_1\ \cdots\ x_{q-1})$, $x_i \in [2^c]$, we construct a vector of $q+r$ *derived characters* $\vec{z} = (z_0\, z_1\ \cdots\ z_{q+r-1})$, $z_j \in [p]$, $p \geq \max\{2^c, q+r\}$. Some of the derived characters may be input characters.
2. We will have $q+r$ independent tabulated hash functions h_j into $[2^\ell]$, and the hash value is then

   $$h(\vec{x}) = h_0[z_0] \oplus \cdots \oplus h_{q+r-1}[z_{q+r-1}] \tag{2.3}$$

   The domain of the different derived characters depends on the application. Here we just assume that h_j has an entry for each possible value of z_j.

We will now define the notion of a "derived key matrix" along with some simple lemmas. Consider $k' \leq k$ distinct keys $\vec{x}_i = (x_{i,0}\ x_{i,1}\ \cdots\ x_{i,q-1})$, $i \in [k']$, and let the derived characters $\vec{z}_i$ be $(z_{i,0}\ z_{i,1}\ \cdots\ z_{i,q+r-1})$. We then define the *derived key matrix* as

$$D = \begin{bmatrix} z_{0,0} & z_{0,1} & \cdots & z_{0,q+r-1} \\ z_{1,0} & z_{1,1} & \cdots & z_{1,q+r-1} \\ & & \ddots & \\ z_{k'-1,0} & z_{k'-1,1} & \cdots & z_{k'-1,q+r-1} \end{bmatrix}$$

LEMMA 2.1. *Suppose for any $k' \leq k$ distinct keys $\vec{x}_i$, $i \in [k']$, the derived key matrix D contains some element that is unique in its column, then the combined hash function h defined in (2.3) is k-universal if all the h_j, $j \in [q+r]$, are independent k-universal hash functions.*

Proof. Consider a set of k distinct keys along with their derived key matrix D. For any set of k hash values v_i, $i \in [k]$, we have to show that

$$\Pr\{h(\vec{x}_i) = v_i,\ \forall i \in [k])\} = 1/2^{k\ell}$$

By assumption, there is an element z_{i_0,j_0} that is unique in column j_0. Since the h_i are independent k-universal hash functions, each character in each column is hashed independently. Without loss of generality, we can assume that the hash value of z_{i_0,j_0} is picked last. When all the other characters are hashed, we obtain hash values for each of the other $k-1$ keys. By induction, these are hashed $(k-1)$-universally, so

$$\Pr\{h(\vec{x}_i) = v_i,\ \forall i \in [k] \setminus \{i_0\}\} = 1/2^{(k-1)\ell}$$

However, z_{i_0,j_0} is hashed independently by h_{j_0}, and

$$\begin{aligned}&\Pr\{h(\vec{x}_{i_0}) = v_{i_0}\}\\ &= \Pr\left\{h_{i_0}[z_{i_0,j_0}] = v_{i_0} \oplus \bigoplus_{i \neq i_0} h_i[z_{i,j_0}]\right\} = 1/2^{\ell}\end{aligned}$$

Hence, $\Pr\{h(\vec{x}_i) = v_i,\ \forall i \in [k])\} = 1/2^{k\ell}$, as desired.

In our constructions for 4-universal hashing, the input characters will all be used as derived characters, and then we can simplify the assumption of Lemma 2.1 to dealing with exactly 4 keys.

LEMMA 2.2. *Suppose all input characters are used as derived characters and that for any* 4 *distinct keys* $\vec{x}_i$, $i \in [4]$, *the derived key matrix D contains some element that is unique in its column, then the combined hash function h defined in (2.3) is* 4*-universal if all the* h_j, $j \in [q+r]$, *are independent* 4*-universal hash functions.*

Proof. Consider $k' < 4$ distinct keys. The distinctness implies that some column of input characters in D has at least two different elements, and for $k' < 4$, one of these elements must be unique in its column. Hence, for $k = 4$, the condition of Lemma 2.1 is satisfied if it is satisfied for $k' = k = 4$.

2.2 4-universal hashing with two characters.

THEOREM 2.1. *If keys are divided into 2 characters, then*

$$h(xy) = h_0[x] \oplus h_1[y] \oplus h_2[x+y]$$

is a 4*-universal hash function if* h_0, h_1, *and* h_2 *are independent* 4*-universal hash functions into* $[2^\ell]$.

Proof. For any 4 distinct keys $x_i\, y_i$, $i \in [4]$, let $z_i = x_i + y_i$. By Lemma 2.2, it suffices to show that the derived key matrix

$$D = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}$$

has an element that is unique in its column. Without loss of generality, we may assume that each element appears at least twice in the input columns $\{x_j\}$ and $\{y_j\}$. Since the four keys $x_i\ y_i$ are distinct and D has only four rows, it is easy to see that each x_i and y_i must appear exactly twice in its column. Without loss of generality, we can assume the four distinct keys $x_i\, y_i$ are a_0b_0, a_0b_1, a_1b_0, and a_1b_1, where $a_0 < a_1$ and $b_0 < b_1$. This implies $a_0 + b_0 < a_0 + b_1 < a_1 + b_1$ and $a_0 + b_0 < a_1 + b_0 < a_1 + b_1$. So both $a_0 + b_0$ and $a_1 + b_1$ are unique in column $\{z_j\}$.

A slight caveat of the above scheme is that $x+y$ requires one more bit than x and y, hence that h_2 needs to be over a domain that is twice as large. It would have been nicer if we could just apply h_2 to $x \oplus y$ instead of $x+y$, but then the combined function guarantees $h(00) \oplus h(01) \oplus h(10) \oplus h(11) \equiv 0$, and is therefore not 4-universal.

The following theorem shows that we can still achieve 4-universal hashing if we replace $x + y$ with addition over an *odd* prime field $\mathbb{Z}_p$ containing the domain for input characters. With concrete character length $c = 8, 16$, we can exploit that $2^c + 1$ prime. Then the domain of the derived character $x + y$ is only one bigger than that of the input characters.

THEOREM 2.2. *Theorem 2.1 still holds if we perform addition over an odd prime field* $\mathbb{Z}_p$ *containing the domain for input characters. That is, if keys are divided into 2 characters in* $[2^c]$, *then*

$$h(xy) = h_0[x] \oplus h_1[y] \oplus h_2[x + y \mod p]$$

is a 4*-universal hash function, where* $p > 2^c$ *is an odd prime and* h_0, h_1, *and* h_2 *are independent* 4*-universal hash functions into* $[2^\ell]$.

Proof. For this proof, we use '$+$' and '$-$' to denote addition and subtraction over $\mathbb{Z}_p$. As in the proof of Theorem 2.1, we may assume that the 4 distinct keys $x_i y_i$ are a_0b_0, a_0b_1, a_1b_0, and a_1b_1, respectively, where $a_0 \neq a_1$ and $b_0 \neq b_1$. With $z_i = x_i + y_i$, we need to show that some z_i is unique. Clearly, $z_0 = a_0 + b_0 \neq a_0 + b_1 = z_1$. Similarly, $z_0 \neq z_2$. Hence, if z_0 is not unique, $z_0 = z_4$, and if z_2 is not unique, $z_2 = z_3$. But these equalities imply $a_0 + b_0 + a_0 + b_1 = a_1 + b_0 + a_1 + b_1$, or $a_0 + a_0 = a_1 + a_1$. Therefore, there exists an element $e = a_0 - a_1 \neq 0$ such that $e + e = 0$. This is impossible in an odd prime field, so we conclude that some z_i is unique.

Note that if we perform addition on any even field over $[2^c]$, the combined hash function $h(xy) = h_0[x] \oplus h_1[y] \oplus h_2[x + y]$ is not 4-universal. This is because for any even field over $[2^c]$, there always exists an element $e \neq 0$ such that $e + e = 0$, where 0 is the zero element of the field. This means $h(00) \oplus h(0\,e) \oplus h(e\,0) \oplus h(e\,e) \equiv 0$. Therefore, h cannot be 4-universal.

Interestingly, however, $h(xy) = h_0[x] + h_1[y] + h_2[x \oplus y]$ is indeed 4-universal if h_0, h_1, and h_2 are 4-universal hash functions into $[p']$, where p' is an odd prime, and addition is performed over $\mathbb{Z}_{p'}$. But since in practice it is often much more convenient to deal with bit strings, we will not go into details on this.

2.3 4-universal hashing with q characters. For 4-universal hashing with more than two input characters, we can recursively apply the two-character scheme, but then, for q characters, we would end up using $q^{\log_2 3}$ hash functions. Here we show that we can get down to $2q - 1$.

Let $r = q - 1$. Given q input characters $\vec{x} = (x_0\ x_1\ \cdots\ x_{q-1})$, $x_i \in [2^c]$, we obtain $q + r$ characters by including the q input characters themselves together with r additional characters $\vec{y} = (y_0\ y_1\ \cdots\ y_{r-1})$ derived using

$$\vec{y} = \vec{x}G$$

where G is a $q \times r$ generator matrix with the property that any square submatrix of G has full rank, and vector element additions and multiplications are performed over an *odd* prime field $\mathbb{Z}_p$, $p \geq \max\{2^c, q + r\}$. We then use the above general hashing framework to combine $q + r$ independent tabulated 4-universal hash functions.

For small q, the generator matrix G can be constructed manually. In particular, if $q = 2$ (thus $r = 1$), we can just use $G = (1\ 1)^T$, which gives the scheme in Theorem 2.2. For large q, we can use a $q \times r$ Cauchy matrix over prime field $\mathbb{Z}_p$ (which mandates $p \geq q + r$):

$$\mathcal{C}^{q\times r} = \left[\frac{1}{i+j+1}\right]_{i\in[q],\ j\in[r]}$$
$$= \begin{bmatrix} \frac{1}{0+0+1} & \frac{1}{0+1+1} & \cdots & \frac{1}{0+(r-1)+1} \\ \frac{1}{1+0+1} & \frac{1}{1+1+1} & \cdots & \frac{1}{1+(r-1)+1} \\ \cdots & & \ddots & \cdots \\ \frac{1}{(q-1)+0+1} & \frac{1}{(q-1)+1+1} & \cdots & \frac{1}{(q-1)+(r-1)+1} \end{bmatrix}$$

THEOREM 2.3. *Let G be a $q \times r$ generator matrix with the property that any square submatrix of G has full rank over prime field $\mathbb{Z}_p$, where $p \geq \max\{2^c, q + r\}$ is an odd prime. Given any q characters $\vec{x} = (x_i)$, $i \in [q]$, let $\vec{y} = (y_j)$, $j \in [r]$, be the $r = q-1$ additional characters derived using $\vec{y} = \vec{x}G$, then the combined hash function*

$$h(\vec{x}) = h_0[x_0]\oplus\cdots\oplus h_{q-1}[x_{q-1}]\oplus\tilde{h}_0[y_0]\oplus\cdots\oplus\tilde{h}_{r-1}[y_{r-1}]$$

is a 4-universal hash function if hash functions h_i ($i \in [q]$) and $\tilde{h}_j$ ($j \in [r]$) are independent 4-universal hash functions into $[2^\ell]$.

Proof. Consider 4 distinct q-character keys $\vec{x}_i = (x_{i,0}\ \cdots\ x_{i,q-1})$, $i \in [4]$. Let $\vec{y}_i = (y_{i,0}\ \cdots\ y_{i,r-1}) = \vec{x}_iG$ and consider the the derived key matrix

$$D = \begin{bmatrix} x_{0,0} & \cdots & x_{0,q-1} & y_{0,0} & \cdots & y_{0,r-1} \\ x_{1,0} & \cdots & x_{1,q-1} & y_{1,0} & \cdots & y_{1,r-1} \\ x_{2,0} & \cdots & x_{2,q-1} & y_{2,0} & \cdots & y_{2,r-1} \\ x_{3,0} & \cdots & x_{3,q-1} & y_{3,0} & \cdots & y_{3,r-1} \end{bmatrix}$$
$$= [\ X\quad Y\ ]$$

where X and Y are the submatrices formed by vectors $\vec{x}_i$ and $\vec{y}_i$, respectively. Clearly, $Y = XG$.

By Lemma 2.2, we only need to prove that some element of D is unique in its column.

Assume by way of contradiction that each element of D appears at least twice in its column. Then each column $D[\cdot, j]$ must be either of type 0: $(a\ a\ a\ a)^T$, in which all 4 elements of the column are equal, or one of the three proper types in which each element appears exactly twice: type α: $(a\ a\ b\ b)^T$, type β: $(a\ b\ a\ b)^T$, and type γ: $(a\ b\ b\ a)^T$.

For $t = 0, \alpha, \beta, \gamma$, let X_t be the possibly empty submatrix of X that consists of all columns of type t. Also, let G_t be the submatrix of G consisting of the rows j such that column j of X is of type t. Finally, we define $Y_t = X_tG_t$. Then $Y = \sum_{t=0,\alpha,\beta,\gamma} Y_t$.

Consider a specific derived column $Y[\cdot, j]$, $j \in [r]$. For $t = 0, \alpha, \beta, \gamma$, $Y_t[\cdot, j]$ is of type 0 or type t as it is a linear combination of input columns of type t. We say type $t \neq 0$ is *present* in derived column $Y[\cdot, j]$ if $Y_t[\cdot, j]$ is of type t.

We will now prove that at most one type can survive in each derived column $Y[\cdot, j]$, $j \in [r]$. Suppose for a contradiction that we have at least two present types. By symmetry, we may assume that α and β are present. Then $Y_\alpha[\cdot, j] = (a_0a_0b_0b_0)^T$ and $Y_\beta[\cdot, j] = (a_1b_1a_1b_1)^T$, so from the proof of Theorem 2.2, we know that $Y_\alpha[\cdot, j] + Y_\beta[\cdot, j]$ has a unique character. If γ is not present, $Y_0[\cdot, j] + Y_\gamma[\cdot, j]$ is of type 0, and then we know that $Y[\cdot, j]$ has a unique character. Otherwise, all three types $\neq 0$ are present and symmetric in that regard. If we don't have a unique character in $Y[\cdot, j]$, it is of some type, say 0 or γ. This implies that $Y_\alpha[\cdot, j] + Y_\beta[\cdot, j] = Y[\cdot, j] - Y_0[\cdot, j] + Y_\gamma[\cdot, j]$ is of type γ contradicting that $Y_\alpha[\cdot, j] + Y_\beta[\cdot, j]$ has a unique character.

Let n_t be the number of columns in X_t. Next, we prove that if $n_t > 0$ and $t \neq 0$, then G has at most $n_t - 1$ derived columns $Y[\cdot, j]$ where type t does not survive. Assume by contradiction that there are n_t or more derived columns where n_t does not survive. We can then find an $n_t \times n_t$ submatrix G_t^0 of G_t consisting of columns j such that type t is not present in derived column $Y[\cdot, j]$. Then, for any two rows $\vec{a}$ and $\vec{b}$ of X_t, $\vec{a}G_t^0 = \vec{b}G_t^0$. However, X_t has different rows so this contradicts the fact that all square submatrices of G have full rank.

From the above results, we know that G cannot have more than $n_0 + \sum_{t=\alpha,\beta,\gamma} \max\{0, n_t - 1\}$. Since the input keys are distinct, there has to be at least two proper types t with $n_t > 0$. Hence $n_0 + \sum_{t=\alpha,\beta,\gamma} \max\{0, n_t - 1\} \leq q-2$. However, G has $r = q - 1$ columns, so this gives us the desired contradiction.

2.3.1 Relaxed and efficient computation of $\vec{x}G$ on $\mathbb{Z}_p$. With the above scheme, we only need $2q - 1$ table lookups to compute the hash value for q input characters. However, to make the scheme useful in practice, we still need to compute $\vec{y} = \vec{x}G$ very efficiently, which requires $O(qr) = O(q^2)$ multiplications and additions on $\mathbb{Z}_p$ using schoolbook implementation. Below we describe several techniques to get down to $O(q)$ time.

Multiplication through tabulation. Let $\vec{G}_i$. $i \in [q]$, be the q rows of the generator matrix G from Theorem 2.3.

Then

$$\vec{y} = \vec{x}\,G = (x_0\ \cdots\ x_{q-1}) \begin{bmatrix} \vec{G}_0 \\ \vdots \\ \vec{G}_{q-1} \end{bmatrix} = \sum_{i\in[q]} x_i\,\vec{G}_i$$

Therefore, we can avoid all the multiplications by storing with each x_i, not only $h_i[x_i]$, but also the above vector $x_i\,\vec{G}_i$, denoted $\vec{g}_i(x_i)$. Then we compute $\vec{y}$ as the sum $\sum_{i\in[q]} \vec{g}_i(x_i)$ of these tabulated vectors.

Using regular addition. We will now argue that for 4-universality, it suffices to compute $\sum_{i\in[q]} \vec{g}_i(x_i)$ using regular integer addition rather than addition over $\mathbb{Z}_p$. What was shown in the proof of Theorem 2.3 was that some element of the derived key matrix D was unique in its column. However, all elements were from $[p]$ so the uniqueness cannot be destroyed by adding a variable multiples of p to the elements, but this is exactly the effect of using regular integer addition rather than addition over $\mathbb{Z}_p$.

Parallel additions. To make the additions efficient, we can exploit bit-level parallelism by packing the $\vec{g}_i(x_i)$ into bit-strings with $\lceil \log_2 q \rceil$ bits between adjacent elements. Then we can add the vectors by adding the bit strings as regular integers. By Bertand's Postulate, we can assume $p < 2^{c+1}$, hence that each element of $\vec{g}_i(x_i)$ uses $c+1$ bits. Consequently, we use $c' = c + 1 + \lceil \log_2 q \rceil$ bits per element.

For any application we are interested in, $1 + \lceil \log_2 q \rceil \leq c$, and then $c' \leq 2c$. This means that our vectors are coded in bit-strings that are at most twice as long as the input keys. We have assumed our input keys are contained in a word. Hence, we can perform each vector addition with two word additions. Consequently, we can perform all $q-1$ vector additions in $O(q)$ time.

In our main tests, things are even better, for we use 16-bit characters of single and double words. For single words of 32 bits, this is the special case of two characters. For double words of 64 bits, we have $q = 4$ and $r = q - 1 = 3$. This means that the vectors $\vec{g}_i(x_i)$ are contained in integers of $rc' = 3(16 + 1 + 2) = 57$ bits, that is, in double words. Consequently, we can compute $\sum_{i\in[q]} \vec{g}_i(x_i)$ using 3 regular double word additions.

Compression. With regular addition in $\sum_{i\in[q]} \vec{g}_i(x_i)$, the derived characters may end up as large as $q(p-1)$, which means that tables for derived characters need this many entries. If memory becomes a problem, we could perform the mod p operation on the derived characters after we have done all the additions, thus placing the derived characters in $[p]$. This can be done in $O(\log q)$ total time using bit-level parallelism like in the vector additions.

However, for character lengths $c = 8, 16$, we can do even better exploiting that $p = 2^c + 1$ is a prime. We are then going to place the derived characters in $[2^c + q]$. Consider a vector element $a < qp$. Let $a' = a \wedge (2^c - 1) + q - (x \gg c) \wedge (2^c - 1)$. Here $\gg$ denotes a right shift and $\wedge$ denotes bit-wise AND. Then it is easy to show that $0 \leq y < 2^c + q$ and $a' \equiv a + q \pmod{p}$. Adding q and a variable multiple of p to each element of the derived key matrix does not destroy the uniqueness of an element in a column, so our hash function remains 4-universal with these compressed derived characters. The transformation from a to a' can be performed in parallel for a vector of derived characters. With appropriate precomputed constants, the compression is performed in 5 C-level instructions.

2.4 k-universal hashing with q characters.

Here we present a scheme for general k-universal hashing using $(k-1)(q-1)+1$ k-universal hash functions. For $k = 4$, this is not as good as the previous results, but it does have the advantage that it allows the derived characters to have the same length as the input characters.

Let $r = (k-2)(q-1)$. Given q input characters $\vec{x} = (x_i)$ $(i \in [q])$, we derive $q + r = (k-1)(q-1)+1$ characters $\vec{z} = (z_j)$ $(j \in [q+r])$ using

$$\vec{z} = \vec{x}\,H$$

where H is a $q \times (q+r)$ generator matrix with the property that any $q \times q$ submatrix of H has full rank on a (possibly even) field with size $\geq \max\{2^c, q+r\}$. Matrices with this property are commonly used in coding theory to generate *erasure resilient codes* [3, 14]. For example, we can choose $H = [I_q\ G]$ where where I_q is the $q \times q$ identity matrix and G is a $q \times r$ matrix with full rank of all square submatrices. Then we get a scheme just like in §2.3 where G can be the Cauchy matrix $\mathcal{C}^{q\times r}$. However, this time we may use an even field such as $\mathbb{GF}(2^c)$, and that gives us some substantial savings to be explored later. Another possible choice of H is a $q \times (q+r)$ Vandermonde matrix:

$$\mathcal{V}^{q\times(q+r)} = \left[j^i\right]_{i\in[q],\ j\in[q+r]}$$
$$= \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0^1 & 1^1 & 2^1 & \cdots & (q+r-1)^1 \\ \cdots & & & \ddots & \cdots \\ 0^{q-1} & 1^{q-1} & 2^{q-1} & \cdots & (q+r-1)^{q-1} \end{bmatrix}$$

THEOREM 2.4. *Let H be a $q \times (q+r)$ matrix with the property that any $q \times q$ submatrix of H has full rank on a (possibly even) field with size $\geq \max\{2^c, q+r\}$, where c is the length of the input characters. For any given q characters $\vec{x} = (x_i)$, $i \in [q]$, derive $q+r = (k-1)(q-1)+1$ characters $\vec{z} = (z_j)$, $j \in [q+r]$. Then the combined hash function*

$$h(\vec{x}) = h_0[z_0] \oplus \cdots \oplus h_{q+r-1}[z_{q+r-1}]$$

is a k-universal hash function if h_i $(i \in [q+r])$ are independent tabulated k-universal hash functions into $[2^\ell]$.

Proof. By Lemma 2.1, we only need to prove for any $k' \leq k$ distinct keys $\vec{x}_i$, $i \in [k']$, some element of the derived key matrix D is unique in its column. Suppose for a contradiction that every element appears at least twice in its column. For each column $D[\cdot, j]$, define $E_j = \{(a,b) \mid a, b \in [k'] \text{ s.t. } a < b \wedge D[a,j] = D[b,j]\}$. Then we have $|E_j| \geq k'/2$. As the result, $\sum_{j\in[q+r]} |E_j| \geq$

$(q+r)\cdot k'/2 = ((k-1)(q-1)+1)\cdot k'/2 > (q-1)\binom{k'}{2}$. But there are only $\binom{k'}{2}$ different (a,b) pairs ($a,b \in [k']$, $a<b$). Therefore, there must exist a pair (a_0,b_0) that appears in at least q different E_j, $j \in [q+r]$.

Let D' and H' be the $q\times q$ submatrices of D and H consisting of columns j with $(a_0,b_0)\in E_j$. In D', rows a_0 and b_0 are identical, so $\vec{x}_{a_0}H' = \vec{x}_{b_0}H'$. However, $\vec{x}_{a_0} \neq \vec{x}_{b_0}$, so this contradicts the fact that H' is a $q\times q$ matrix with full rank.

2.4.1 Efficient computation of $\vec{z}$ over $\mathbb{GF}(2^c)$.

As mentioned above, we can pick $H = [I_q\ G]$ and thus get a scheme like in §2.3 with $\vec{z}$ being the concatenation of $\vec{x}$ and $\vec{y} = \vec{x}G$. With the notation of §2.3.1, we compute $\vec{y}$ as the sum $\sum_{i\in[q]} \vec{g}_i(x_i)$ of the tabulated vectors $\vec{g}_i(x_i)$. However, this time, we may work in the even field $\mathbb{GF}(2^c)$. Consequently, the elements of $g_i(x_i)$ are in $[2^c]$ and addition over $\mathbb{GF}(2^c)$ is just $\oplus$ as supported directly in C without any need for carry bits. Thus, each vector $\vec{g}_i(x_i)$ is represented as a bit-string of length $rc=(k-2)(q-1)c$, and we just need to $\oplus$ these vectors to produce $\vec{y}$. The resulting derived characters are all in $[2^c]$. This scheme is thus in many ways simpler than our specialized scheme for 4-universal hashing over $\mathbb{Z}_p$. However, for $k=4$, our general scheme performs worse because it uses $2(q-1)$ derived characters, whereas our specialized scheme only uses $q-1$ derived characters. So far, we do not understand if this is an inherent advantage of $\mathbb{Z}_p$ over $\mathbb{GF}(2^c)$, or if there is a smarter way of exploiting $\mathbb{GF}(2^c)$.

3 Performance evaluation.

We have implemented our schemes and *CW-trick* in C. Table 1 compares the different algorithms both in terms of C-level instruction count and actual running times on two machines with different architecture and operating systems. OldTable is simply the standard 2-universal hashing mentioned in §1.2 obtained by hashing each character independently using 16-bit characters. This is currently the fastest known method for 2-universal hashing, hence an interesting benchmark.

CWtrick61 and CWtrick89 are CW-trick schemes as described in §1.4 with Mersenne primes $2^{61}-1$ and $2^{89}-1$, respectively. The actual code for CWtrick61 is found in §5.3 while the code for CWtrick89 is deferred to [18, §B.5]. The code for CWtrick89 gains speed from only producing the 64 least significant bits of the result assuming that we only need that many hashed bits.

Table and CompressTable are instances of our new tabulation based 4-universal hashing schemes from §2.3 with 16-bit input characters. With Table the derived characters are not compressed, and may be as large as $q\times 2^{16}$ with $q=2,4$ depending on whether the input is 32 or 64 bits. With CompressTable, the derived characters are less than $2^{16}+q$, so they need much smaller tables. The actual codes for 32-bit keys is found in §5.2. The code for 64-bit keys is deferred to [18, §B.3].

The C-level instruction count can be read directly from the code and is thus independent of compiler and machine architecture. We see that Table gains more than a factor 5 over CW-trick, both for 32 and 64 bit keys.

When it comes to actual running time, we see that the C-level instruction count gives a good rough estimate of the relative performances, yet there is a glaring contrast on Computer A between Table and CompressTable on 64 bit keys. In this case, Table uses roughly 4 times as much space as CompressTable, so it is natural to attribute its slowness to use of slower memory. Similarly, the relative slowness of CW-trick can be attributed to its use of multiplication. All in all, for running times, we see that CompressTable consistently wins by a factor of 5 over CW-trick. Table is even faster in most cases when memory is not a problem. Even when memory starts to become a problem (Table with 64 bit keys on Computer A), it is still more than 4 times faster than CW-trick.

Summing up, we have shown that tabulation can be used for k-universal hashing, and for the important case of 4-universal hashing, we have gained a factor of 5 in speed over the previous fastest methods, making it a much more viable method for time critical streaming applications.

4 Second moment estimation.

Let $\mathcal{S} = (a_1,w_1),(a_2,w_2),...,(a_s,w_s)$ be a data stream, where each key a_i is a member of $[u]$. Let $v_a = \sum_{i:\ a_i=a} w_i$ be the total weights associated with key $a\in[u]$. Define, for each $j\ge 0$,

$$F_j = \sum_{a\in[u]} v_a^j$$

The second moment, F_2, is of particular interest, since it arises in various applications.

4.1 Second moment estimators.

Classic estimator. The classic method for estimating F_2 by Alon *et. al.* [1] uses n counters c_i ($i\in[n]$) and n independent 4-universal hash functions s_i that map $[u]$ into $\{-1,1\}$. When a new data item (a,w) arrives, all n counters are updated using $c_i{+}{=}\ s_i(a)\cdot w$ ($i\in[n]$). F_2 is then estimated as $X_{\text{classic}} = \sum_{i\in[n]} c_i^2/n$. Following the analysis in [1], we have $\mathrm{E}[X_{\text{classic}}] = F_2$ and $\mathrm{Var}[X_{\text{classic}}] = \sum_{a\neq b} 2\,v_a^2 v_b^2 = 2(F_2^2 - F_4)/n$.

Count sketch based estimator. Recently, Charikar *et. al.* [5] described a data structure called *count sketch* for keeping track of frequent items in a data stream. We can adapt count sketch to make second moment estimation. Using this method, we need n counters c_i ($i\in[n]$) and two independent 4-universal hash functions $h:[u]\to[n]$ and $s:[u]\to\{-1,1\}$. When a new data item (a,w) arrives, a single counter $c_{h(a)}$ is updated using $c_{h(a)}{+}{=}\ s(a)\cdot w$. F_2 is then estimated as $X_{\text{count\_sketch}} = \sum_{i\in[n]} c_i^2$. We can prove that $\mathrm{E}[X_{\text{count\_sketch}}] = F_2$ and $\mathrm{Var}[X_{\text{count\_sketch}}] = 2(F_2^2-F_4)/n$. Therefore, $X_{\text{count\_sketch}}$ achieves the same variance as X_{classic} with substantially lower update cost.

Fast count sketch based estimator. An alternative way of implementing the count sketch scheme is to use $2n$ counters c_i ($i\in[2n]$) and a 4-universal hash function $h:[u]\to$

k-universal	key bits $\to$ hash bits	algorithm	C-level instructions	running time (sec)	
				computer A	computer B
2	$32 \to 64$	OldTable32	5	0.17	0.31
4	$32 \to 61$	CWtrick61	41	1.82	2.95
4	$32 \to 64$	Table32	8	0.30	0.55
4	$32 \to 64$	CompressTable32	12	0.34	0.55
2	$64 \to 64$	OldTable64	11	0.35	0.48
4	$64 \to 64$	CWtrick89	184	6.83	12.31
4	$64 \to 64$	Table64	32	1.56	2.22
4	$64 \to 64$	CompressTable64	37	1.04	2.53
Update 2nd moment in stream			11	0.50	0.68

Table 1: C-level instruction count plus running times for performing 10 million hash computations on computer A (400 MHz SGI R12k processor running IRIX64 6.5) and B (900 MHz Ultrasparc-III processor running Solaris 5.8).

$[2n]$. When a new data item (a, w) arrives, w is directly added to the counter $c_{h(a)}$: $c_{h(a)} += w$. In the end F_2 is estimated using the alternating sum $X_{\text{fast\_count\_sketch}} = \sum_{i\in[n]}(c_{2i} - c_{2i+1})^2$. $X_{\text{fast\_count\_sketch}}$ achieves the same variance as $X_{\text{count\_sketch}}$, but is faster because the direct update of a counter based on a single hash value is much simpler. However, such simplicity comes at the cost of doubling the space.

Our new estimator. Here we present a new estimator that achieves the same speed and variance as the fast count sketch based estimator without having to double the space. Instead of using $2n$ counters, our new method uses $m = n + 1$ counters c_i ($i \in [m]$), and a 4-universal hash function $h : [u] \to [m]$. The update algorithm is exactly the same as that of the fast count sketch based estimator: when a new data item (a, w) arrives, w is added to the counter $c_{h(a)}$: $c_{h(a)} += w$. But the estimation formula is quite different. We use

$$X_{\text{new}} = \frac{m}{m-1}\sum_{i\in[m]} c_i^2 - \frac{1}{m-1}(\sum_{i\in[m]} c_i)^2$$

Note that we do not worry about the cost of adding up counters done in the end. Hence, it is not considered a problem to have a more complex sum for this. In [18, §A], we prove

THEOREM 4.1. *If h is 2-universal,* $\mathrm{E}[X_{\text{new}}] = F_2$. *If h is 4-universal,* $\mathrm{Var}[X_{\text{new}}] = 2(F_2^2 - F_4)/(m-1) = 2(F_2^2 - F_4)/n$.

4.2 Performance evaluation. The code for second moment estimation can be found in §5.4. We used $m = 2^{15}$, giving us a relative standard error below $\sqrt{2/(m-1)} \approx 2^{-7} < 1\%$. The last line in Table 1 shows the instruction count and the running times for performing 10 million hash computation and counter updates. This should be compared with the hash computation alone in Table32. We see that the additional overhead is limited. Even in the worst case when all packets are of the minimum IP packet size of 40 bytes (320 bits), the counter update part can easily keep up with $320 \times 10^7/0.68 = 4.7 \times 10^9$ bits per second on the slower computer, which is nearly twice as fast as OC48 speed (2.48 Gbps). With enough buffering, if the average IP packet size is above 85 bytes (680 bits), which is generally the case in today's Internet, we can even keep up with OC192 speed (10 Gbps).

5 Code.

In this section, we present some of the code discussed in this paper, justifying the concrete C-level instruction counts. The remaining code is found in [18, §B].

5.1 Common data types and macros.

```
typedef unsigned int        INT32;
typedef unsigned long long  INT64;

const INT64 LowOnes = (((INT64)1)<<32)-1;
const INT32 HalfLowOnes = (((INT32)1)<<16)-1;

#define LOW32of64(x)   ((x)&LowOnes)
#define HIGH32of64(x)  ((x)>>32)
#define LOW16of32(x)   ((x)&HalfLowOnes)
#define HIGH16of32(x)  ((x)>>16)
```

5.2 Tabulation based hashing for 32-bit keys using 16-bit characters.

```
/* tabulation based hashing for 32-bit key x
 * using 16-bit characters. T0, T1, T2 are
 * precomputated tables */
inline INT64 Table32(INT32 x, INT64 T0[], INT64
                     T1[],INT64 T2[]){
  INT32 x0, x1, x2;
  x0 = LOW16of32(x);
  x1 = HIGH16of32(x);
  x2 = x0 + x1;
  x2 = compress32(x2); //optional compression
  return T0[x0] ^ T1[x1] ^ T2[x2];
} // 8 + 4 = 12 instructions

/* optional compression */
inline INT32 compress32(INT32 i) {
  return 2 - HIGH16of32(i) + LOW16of32(i);
} // 4 instructions
```

The code uses 12 instructions (8 without compression), including 3 lookups.

5.3 CW Trick for 32-bit keys with prime $2^{61} - 1$.

```
const INT64 Prime = (((INT64)1)<<61) - 1;

/* Computes ax+b mod Prime, possibly +2*Prime,
 * exploiting the structure of Prime.*/
inline INT64 MultAddPrime(INT32 x, INT64 a,
                          INT64 b) {
  INT64 a0,a1,c0,c1,c;
  a0 = LOW32of64(a)*x;
  a1 = HIGH32of64(a)*x;
  c0 = a0+(a1<<32);
  c1 = (a0>>32)+a1;
  c  = (c0&Prime)+(c1>>29)+b;
  return c;
} // 12 instructions

/* CWtrick for 32-bit key x with
 * prime 2^61-1 */
inline INT64 CWtrick(INT32 x, INT64 A,
                     INT64 B, INT64 C,
                     INT64 D) {
  INT64 h;
  h = MultAddPrime(MultAddPrime(
       MultAddPrime(x,A,B),x,C),x,D);
  h = (h&Prime)+(h>>61);
  if (h>=Prime) h-=Prime;
  return h;
} // 12*3 + 5 = 41 instructions
```

The code uses 41 instructions including 6 multiplications.

5.4 Second moment estimation.

```
#define NumCounters 32768       // (1<<15)
INT64 Counters[NumCounters];

/* precomputed tables whose hash strings
 *  only use 15 least significant bits */
INT64 *T0, *T1, *T2;

inline void StreamUpdate2nd(INT32 ipaddr,
                            INT32 size) {
  Counters[Table32(ipaddr,T0,T1,T2)]+=size;
} // 3 instructions plus those in Table32.

double StreamEstimate2nd() {
  int i;
  INT64 c;
  double sum = 0, sqsum = 0;
  for (i = 0; i < NumCounters; i++) {
    c = Counters[i];
    sum += c;
    sqsum += c*c;
  }
  return sqsum + (sqsum-sum*sum)/
                  (NumCounters-1);
}
```

The code for updating the counters adds 3 instructions to those in Table32, including one additional lookup.

Acknowledgment. We would like to thank Martin Dietzfelbinger and Muthu(krishnan) for very useful comments and suggestions.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. System Sci.*, 58(1):137–147, 1999.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, 2002.

[3] J. Bloemer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, ICSI, Berkeley, California, Aug. 1995. http://www.icsi.berkeley.edu/˜luby/PAPERS/cauchypap.ps.

[4] J. Carter and M. Wegman. Universal classes of hash functions. *J. Comp. Syst. Sci.*, 18:143–154, 1979.

[5] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. 29th ICALP, LNCS 2380*, pages 693–703, 2002.

[6] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proc. 13th STACS, LNCS 1046*, pages 569–580, 1996.

[7] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proc. 19th ICALP*, LNCS 623, pages 235–246, 1992.

[8] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25:19–51, 1997.

[9] M. Dietzfelbinger and P. Woelfel. Almost random graphs with simple hash functions. In *Proc. 35th STOC*, pages 629–638, 2003.

[10] Intel. The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference (Order Number 245471), 2001. http://developer.intel.com/design/pentium4/manuals/245471.htm.

[11] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Proc. ACM Internet Measurement Conference (IMC-03)*, 2003.

[12] Y. Mansour, N. Nisan, and P. Tiwari. The computational complexity of universal hashing. *Theor. Comp. Sc.*, 107:121–133, 1993.

[13] S. Muthukrishnan. Data streams: Algorithms and applications, 2003. Manuscript based on invited talk from *14th SODA*. Available from http://www.cs.rutgers.edu/˜muthu/stream-1-1.ps.

[14] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2):24–36, Apr. 1997.

[15] A. Schönhage and V. Strassen. Schnelle muliplication großer zahlen. *Computing*, 7:281–292, 1971.

[16] M. Thorup. Even strongly universal hashing is pretty fast. In *Proc. 11th SODA*, pages 496–497, 2000.

[17] M. Thorup. Combinatorial power in multimedia processors, 2003. http://www.research.att.com/˜mthorup/PAPERS/multimedia.ps.

[18] M. Thorup and Y. Zhang. Appendix for "Tabulation Based 4-Universal Hashing with Applications to Second Moment Estimation", 2003. http://www.research.att.com/˜mthorup/PAPERS/k-univ-app.ps.

[19] M. Wegman and J. Carter. New hash functions and their use in authentication and set equality. *J. Comp. Syst. Sci.*, 22:265–279, 1981.