Tabulation Based 5-Universal Hashing and Linear Probing

Mikkel Thorup
AT&T Labs—Research
180 Park Avenue
Florham Park, NJ 07932, USA
mthorup@research.att.com

Yin Zhang
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712, USA
yzhang@cs.utexas.edu

Abstract

Previously [SODA'04] we devised the fastest known algorithm for 4-universal hashing. The hashing was based on small pre-computed 4-universal tables. This led to a five-fold improvement in speed over direct methods based on degree 3 polynomials.

In this paper, we show that if the pre-computed tables are made 5-universal, then the hash value becomes 5-universal without any other change to the computation. Relatively this leads to even bigger gains since the direct methods for 5-universal hashing use degree 4 polynomials. Experimentally, we find that our method can gain up to an order of magnitude in speed over direct 5-universal hashing.

Some of the most popular randomized algorithms have been proved to have the desired expected running time using 5-universal hashing, *e.g.*, a non-recursive variant of quicksort takes $O(n \log n)$ expected time [Karloff Raghavan JACM'93], and linear probing does updates and searches in $O(1)$ expected time [Pagh *et al.* SICOMP'09]. In contrast, inputs have been constructed leading to much worse expected performance with some of the classic primality based 2-universal hashing schemes.

In the context of linear probing, we compare our new fast 5-universal hashing experimentally with the fastest known plain universal hashing. We know that any reasonable hashing scheme will work on random input, but from Pagh *et al.*, we know that 5-universal hashing leads to good expected performance on all input. We use a dense interval as an example of a structured yet realistic input, wanting to see if this could push the fastest multiplication-shift based plain universal hashing into bad performance. Even though our 5-universal hashing itself is slower than the fast plain universal hashing, it makes linear probing much more robust.

1 Introduction.

This paper is about efficient 5-universal hashing. For any $n \in \mathbb{N}$, let $[n] = \{0, 1, \cdots, n-1\}$. As defined in [16], 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. We will often contrast k-universal hashing with *(plain) universal hashing* that just requires low collision probability, that is, for any different $x, y \in [n]$, $\Pr_{h \in \mathcal{H}}\{h(x) = h(y)\} \leq 1/m$.

We develop a fast implementation of 5-universal hashing, gaining up to an order of magnitude in speed over direct methods. 5-universal hashing is important because popular randomized algorithms such as linear probing [11] have provably good expected performance with 5-universal hashing. The same holds for a certain non-recursive variant of quicksort [6].

Our new implementation of 5-universal hashing is based on our previous fast scheme for 4-universal hashing [15]. This scheme used some small pre-computed tables. What we show here is that to get 5-universal hashing, we only need to make the pre-computed tables 5-universal. The procedure that computes the hash function is not affected, hence neither is the speed.

We conduct experiments evaluating the speed of our new hash function against alternatives. We also run experiments with linear probing on clustered inputs where we can clearly see the advantages of 5-universal hashing over the fastest multiplication-shift based plain universal hashing. The plain universal hashing is in itself faster, but it sometimes results in far more probes, making the overall process slower and less reliable than our 5-universal scheme.

1.1 k-universal hashing. We will describe in more detail the relevant known methods for k-universal hashing, and show how our new tabulation based 5-universal hashing fits in. We note that for the more complex hash functions with $k > 2$, we will rarely need to hash keys with more than 64 bits, because assuming that the number of keys is $n \ll 2^{32}$, then we can first use plain universal hashing into a 64-bit domain and this mapping is collision-free with high probability. In fact, for our primary application of linear probing, it is shown in [14] that we can first use plain universal hashing into a domain of size n, and then we only need to handle 32-bit keys. Based on this, in the the rest of the paper, we will focus on the hashing of 32 and 64 bit keys in our comparison between different hashing scheme. In fact, our scheme would only look better if we studied 96 or 128 bit integers, but here we focus on the cases that we expect to be most important in practice.

1.1.1 Direct methods.

The classic implementation of k-universal hashing from [16] is to use a degree $k-1$ polynomial over some prime field:

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

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 '$\bmod\ p$' is slow. However, as pointed out in [1], we can get a fast implementation using shifts and bit-wise Boolean operations if p is a so-called Mersenne prime of the form $2^i - 1$. 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 $p = 2^{89} - 1$.

Multiplication-shift based hashing with small universality. For the special case of 2-universal hashing, we have a much faster and more practical method from [2]. If we are hashing from ℓ_{in} to ℓ_{out} bits, for some $\ell \geq \ell_{in} + \ell_{out} - 1$, we pick two random numbers $a, b \in [2^\ell]$, and use the hash function $h_{a,b}$ defined by

$$h_{a,b}(x) = (ax + b) \gg (\ell - \ell_{out}).$$

Here $\gg$ denotes a right shift. The multiplication discards overflow beyond the ℓ bits, as is done automatically in most programming languages if, say, ℓ is 32 or 64. Some generalizations to k-universal hashing for $k > 2$ are also presented in [2], but they would not be faster than the classic method from (1.2).

In fact, if we are satisfied with plain universal hashing, then as shown in [3], it suffices that $\ell \geq \ell_{in}$ and to pick a single odd random number $a \in [2^\ell]$. We then use the hash function h_a defined by

$$h_a(x) = (ax) \gg (\ell - \ell_{out}).$$

As a typical example, if $\ell_{out} \leq \ell_{in} = 32$, then for the 2-universal hashing, we would use a 64 bit number a and 64-bit multiplication. But for plain universal hashing, it suffices to work with 32 bits, which is faster.

The above two schemes can be viewed as instances of multiplicative hashing [8] where the golden ratio of 2^ℓ is recommended as a concrete value of a (with such a fixed value, the schemes lose universality). We refer to them as "multiplication-shift" based methods.

1.1.2 Tabulation based methods.

A totally different way of getting a 2-universal hash value from a key is to divide the key into characters, use an independent tabulated 2-universal function to produce a hash values for each character, and then return the bit-wise exclusive-or of each of these strings. This method goes back to [1]. Theoretically tabulation is incomparable with multiplication based method: we replace multiplication, a comparatively slow operation, with look-up from small tables that should reside in fast memory. An experimental comparison with other methods is found in [13], and the tabulation based approach was found to be faster than other 2-universal methods on most of the computers tested.

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 and we use '[]' around the arguments of the h_i to indicate that the h_i are tabulated so that function values are found by a single array look-up. If $\mathcal{H}$ is 3-universal, then so is h. However, the scheme breaks down above 3-universality. Regardless of the properties of $\mathcal{H}$, $\vec{h}$ is not 4-universal.

In [4, 12, 15] it is shown that we can get higher degrees of universality if we tabulate some extra derived characters. The case where the original key has $q = 2$ characters is particularly nice. It is shown in [15] that

$$h[ab] = h_0[a] \oplus h_1[b] \oplus h_2[a+b] \tag{1.3}$$

is a 4-universal hash function if h_0, h_1, and h_2 are independent 4-universal hash functions. As an example, for 32-bit keys, a and b are 16-bit characters, so the tables h_0 and h_1 are of size 2^{16} while h_2 is of size 2^{17}. This fits quite easily in cache leading to very fast implementations.

For $q > 2$ key characters, it is proved in [15] that we can get 4-universal hashing using $q-1$ extra derived characters, hence $2q-1$ look-ups. The derivation of these extra characters via a Cauchy matrix is a bit complicated to describe but a careful implementation in C from [15] runs fast.

For general $k > 4$, it is shown in [15] that we can get k-universal hashing, first making q look-ups to get $(k-1)(q-1)+1$ derived characters, and then use these as look-ups in k-universal character tables, thus using $k(q-1)+2$ look-ups in total.

The older methods from [4, 12] lead to more look-ups when k is constant, but the method from [12] is better for larger k. Our interest here is 5-universal hashing, and then the current best choice is the method from [15] leading to to $5(q-1)+2$ look-ups.

1.1.3 Our new tabulation based 5-universal hashing.

Our theoretical contribution here is to show that the above

4-universal tabulation scheme from [15] leads to 5-universal hashing as long as the character tables are 5-universal and not just 4-universal. In particular, we get 5-universal hashing with $2q-1$ look-ups.

Proving that we get 5-universal hashing in the 2-character case is quite simple and was noted in [11]. However, for $q>2$, the situation gets complicated. All the previous proofs from [4, 12, 15] of k-universality from derived characters use a peeling lemma of Siegel [12, Lemma 2.6] which identifies a unique character among k keys with derived characters. Here we need a generalized peeling lemma identifying an appropriate full-rank $n \times n$ matrix. The previous unique character forms the special case of a 1×1 matrix.

1.2 Contents. First we describe the previous tabulation based 4-universal hashing from [15] in more detail. Next we give the proof that it also gives 5-universal hashing. Then we switch to experiments, first just looking at the speed of different hashing schemes, second considering the impact on linear probing.

2 Previous tabulation based 4-universal hashing.

In this section, we review our previous tabulation based 4-universal hashing from [15]. In the next section, we show that the same scheme also works for 5-universal hashing.

2.1 General framework. The 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, and those that are not, are called *new*.
2. We will have $q+r$ independent tabulated hash functions h_j into $[2^{\ell_{out}}]$, and the hash value is then

   $$h(\vec{x}) = h_0[z_0] \oplus \cdots \oplus h_{q+r-1}[z_{q+r-1}] \tag{2.4}$$

   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

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

We will use the following "peeling lemma" from [12, Lemma 2.6] (see also [15, Lemma 1]):

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 (2.4) is k-universal if all the h_j, $j \in [q+r]$, are independent k-universal hash functions.*

In [15, Lemma 2], it is noted that

LEMMA 2.2. *Suppose all input characters are used as derived characters, then the unique character condition of Lemma 2.1 is satisfied for any $k' \leq 3$.*

2.2 4-universal hashing with 2 characters.

THEOREM 2.1. *In the case of two-character keys xy, if we use x, y, and $x+y$ as derived characters, then the unique character condition of Lemma 2.1 is satisfied for any $k'=4$. This also holds if '$+$' is in an odd prime field $\mathbb{Z}_p$ containing x and y. In particular,*

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

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

The point in using the above prime field is that it may allow us to reduce the range of $x+y$, hence the size of the table h_2 above. In particular, with 8-bit characters, we can use the prime $p = 2^8+1$ and with 16-bit characters, we can use $p = 2^{16}+1$.

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 $q^{\log_2 3}$ derived characters. As shown in [15] it is show that we can get down to $2q-1$ derived characters.

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 new 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 framework to combine $q+r$ independent tabulated 4-universal hash functions. For example, we can use a $q \times r$ Cauchy matrix below over $\mathbb{Z}_p$ (where $p \geq q+r$):

$$\mathcal{C}^{q\times r} = \left[\frac{1}{i+j+1}\right]_{i\in[q],\ j\in[r]}$$

$$= \begin{bmatrix} \frac{1}{0+0+1} & \frac{1}{0+1+1} & \cdots & \frac{1}{0+(r-1)+1} \\ \frac{1}{1+0+1} & \frac{1}{1+1+1} & \cdots & \frac{1}{1+(r-1)+1} \\ \cdots & & \ddots & \cdots \\ \frac{1}{(q-1)+0+1} & \frac{1}{(q-1)+1+1} & \cdots & \frac{1}{(q-1)+(r-1)+1} \end{bmatrix}$$

THEOREM 2.2. *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 $r = q-1$ new characters derived using $\vec{y} = \vec{x}G$. Then, for any $k' = 4$ distinct keys, one will have a derived character that is unique in its column. Therefore, 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 $[2^\ell]$.

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. In § 3.3, we show efficient techniques to compute $\vec{y} = \vec{x}G$ in $O(q)$ time for general 5-universal hashing.

3 Generalizing to 5-universal hashing.

We will now show that our construction for tabulation-based 4-universal hashing can be directly used to generate 5-universal hashing.

3.1 5-universal hashing with 2 characters. In the case of 2 characters, we note that

LEMMA 3.1. *For any set of $k' = 5$ distinct two-character keys, some character is unique in its column.*

Proof. To get 5 distinct keys, one of the two input columns must contain at least 3 distinct characters. One of these 3 is used at most once in the 5 keys.

This uniqueness for $k' = 5$ is trivial compared with the uniqueness for $k' = 4$ from Theorem 2.1. Further combining with Lemma 2.1 and 2.2, we get

THEOREM 3.1. *In the case of two-character keys xy, the hash function*

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

is a 5-universal hash function if h_0, h_1, and h_2 are independent 5-universal hash functions into $[2^\ell]$.

This simple generalization for the case of two characters was also noted in [11].

3.2 5-universal hashing with q characters. For $q > 2$ characters, we can no longer use the classic peeling lemma (Lemma 2.1). Instead of peeling a unique character, we have to look for a certain full-rank square indicator matrix defined as follows. The unique character is the special case where $n = 1$.

DEFINITION 1. (SPECIAL INDICATOR MATRIX) *From the derived key matrix D, we pick n possibly identical columns $c_0, \cdots, c_{n-1}$, and for each $j \in [n]$, we pick a special character w_j. In the* special indicator matrix *M, each element $M[i,j]$ is a 0/1 indicator telling whether special character w_j appears at row i in column c_j of D. That is,*

$$M[i,j] = \begin{cases} 1, & \text{if } D[i,c_j] = w_j; \\ 0, & \text{otherwise.} \end{cases}$$

DEFINITION 2. (FULL-RANK SQUARE INDICATOR MATRIX) *A special indicator matrix with n columns is considered a* full-rank square indicator matrix *if the following two conditions hold: (i) the indicator matrix has exactly n non-zero rows (*i.e.*, rows with a 1 somewhere), and (ii) these non-zero rows form a $n \times n$ square submatrix that has full rank over $\mathbb{GF}(2)$.*

As a special case, if we have a unique character in some column, then we can make it the only special character, and use it as a 1×1 full-rank square indicator matrix.

As another example, suppose derived key matrix D has 5 rows with the first 3 columns being

$$D[*, 1\text{–}3] = \begin{bmatrix} a & c & e \\ a & d & f \\ a & c & f \\ b & d & e \\ b & d & e \end{bmatrix}.$$

Let $(c_0, c_1, c_2) = (1, 2, 3)$ and $(w_0, w_1, w_2) = (a, c, f)$. The resulting special indicator matrix is

$$M = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

M has exactly 3 non-zero rows: 1, 2, 3. Moreover, the 3×3 submatrix $M[1\text{–}3, *]$ has full-rank on $\mathbb{GF}(2)$. Therefore, M is a full-rank square indicator matrix.

Generalizing Lemma 2.1, we now prove

LEMMA 3.2. *Suppose for any $k' \leq k$ distinct q-character keys $\vec{x_i}$, we can identify a full-rank square indicator matrix in the derived key matrix. Then the combined hash function h defined in (2.4) is k-universal if all the h_j, $j \in [q+r]$, are independent k-universal hash functions.*

Proof. The proof is by induction and is similar to our proof for Lemma 2.1. Consider a set of k distinct keys along with their derived key matrix D. For any 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, we can find $n \geq 1$ (possibly identical) columns $C = (c_0, \cdots, c_{n-1})$ and n special characters $W = (w_0, \cdots, w_{n-1})$ such that (i) the resulting special indicator matrix M has exactly n non-zero rows $R = (r_0, \cdots, r_{n-1})$, and (ii) the $n \times n$ submatrix $M[R, *]$ is full-rank on $\mathbb{GF}(2)$.

Since the h_i are independent k-universal hash functions, each character in each column is hashed independently. Assume w.l.o.g. that the hash values $h_{c_j}[w_j]$ ($j \in [n]$) are picked after all the other characters are hashed. By hashing all the other characters, we obtain hash values for $(k-n)$ keys $h(\vec{x_i})$ ($i \in [k] \setminus R$). By induction, these are hashed $(k-n)$-universally, so

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

Conditioned on the initial hashing of the other characters, we only need to prove

$$\Pr\{h(\vec{x_i}) = v_i, \forall i \in R\} = 1/2^{n\ell}$$

For $\forall i \in R$, let

$$v'_i = v_i \oplus \left(\bigoplus_{j: M[i,j]=0} h_{c_j}[D[i, c_j]] \right) \oplus \left(\bigoplus_{c \notin C} h_c[D[i, c]] \right).$$

Clearly, $h(\vec{x_i}) = v_i, \forall i \in R$ is equivalent to:

$$\bigoplus_j M[i,j] \otimes h_{c_j}[w_j] = v'_i \qquad \forall i \in R \tag{3.5}$$

The fact that $M[R, *]$ is a full-rank square matrix on $\mathbb{GF}(2)$ ensures that given all the v'_i ($i \in R$), there is a unique solution for the hash values $h_{c_j}[w_j]$ ($j \in [n]$). Therefore, the probability for (3.5) to hold is $1/2^{n\ell}$, which completes our proof of Lemma 3.2.

THEOREM 3.2. *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 $r = q - 1$ new characters derived using $\vec{y} = \vec{x}\,G$, then*

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

is a 5-universal hash function if hash functions h_i ($i \in [q]$) and $\tilde{h}_j$ ($j \in [r]$) are independent 5-universal hash functions into $[2^\ell]$.

Proof. Our proof for Theorem 2.2 already establishes that for any $k' \leq 4$ distinct q-character keys $\vec{x_i}$ ($i \in [k']$), we can construct a special indicator matrix with just one column and one 1 in that column (corresponding to the unique element in that column). Therefore, in order to apply Lemma 3.2, we just need to prove that when D has 5 rows and $(2q-1)$ columns, we can always construct a special indicator matrix M with $n \geq 1$ columns and all its non-zero rows form a $n \times n$ full-rank matrix on $\mathbb{GF}(2)$.

Assume that D contains no column with a unique character (otherwise, we are done). Then each column of D will either have one character that appears 5 times, or have two characters that appear 2 times and 3 times, respectively. Let $Minor_D$ be a $0/1$ indicator matrix that indicates whether each element of D is a minority element in its column. Then each column of $Minor_D$ has either five 0s, or two 1s and three 0s. It is easy to see that for any $i_0, i_1 \in \{1, \cdots, 5\}$,

$$Minor_D[i_0, j] = Minor_D[i_1, j] \Longleftrightarrow D[i_0, j] = D[i_1, j].$$

Below we first establish a few lemmas before proving we can always construct a M as required by Theorem 3.2. These lemmas involve the following two conditions:

- (**C1**) no two rows of D share q elements, and
- (**C2**) every column of $Minor_D$ has exactly two 1s and three 0s.

LEMMA 3.3. *Under conditions (C1) and (C2), for any given three rows, say, row 1–3, there are at most $m \triangleq \lfloor (q-1)/2 \rfloor$ distinct columns in $Minor_D$ whose corresponding elements in those three rows are all 0s.*

Proof. Assuming by way of contradiction that $Minor_D$ has $(m+1)$ columns (say, $1, \cdots, m+1$) whose elements in row 1–3 are all 0s. So each of these $(m+1)$ columns in D has equal elements in row 1–3.

Consider the remaining $(2q-2-m)$ columns $m+2, \cdots, 2q-1$. Each of the remaining columns in D has at least two equal elements in row 1–3. There are $\binom{3}{2} = 3$ possible row pairs within $\{1,2,3\}$. Thus there must exist two rows $i_0, i_1 \in \{1,2,3\}$ such that at least $\lceil (2q-2-m)/3 \rceil$ columns have equal elements in row i_0 and row i_1. Since the first $(m+1)$ columns also have equal elements in row i_0 and i_1, the number of columns in which row i_0 and i_1 have equal characters is at least

$$\begin{aligned}
&(m+1) + \lceil (2q-2-m)/3 \rceil \\
&= \left\lfloor \frac{q+1}{2} \right\rfloor + \left\lceil \frac{2q-2-\lfloor (q-1)/2 \rfloor}{3} \right\rceil \\
&= \begin{cases} q/2 + \lceil q/2 - 1/3 \rceil & \text{(if } q \text{ is even)} \\ (q+1)/2 + \lceil \frac{(2q-2)-(q-1)/2}{3} \rceil & \text{(if } q \text{ is odd)} \end{cases} \\
&= q,
\end{aligned}$$

contradicting (C1).

LEMMA 3.4. *Under conditions (C1) and (C2), every row of $Minor_D$ contains at least one* 0*.*

Proof. Assume by way of contradiction that at least one row (say, row 1) of $Minor_D$ contains no 0. That is, all elements of $Minor_D[1, *]$ are 1s. From (C2), each column of $Minor_D$ contains a second 1 in one of four rows: $2, 3, 4, 5$. From Lemma 3.3, for each possible row of the second 1, there are at most $\lfloor (q-1)/2 \rfloor$ distinct columns in D. So altogether the number of columns is at most $4 \times \lfloor (q-1)/2 \rfloor \leq 2 \times (q-1) < 2q-1$, contradicting the fact that D has $2q-1$ columns.

LEMMA 3.5. *Under conditions (C1) and (C2), every row of $Minor_D$ contains at least one* 1*.*

Proof. Assume by way of contradiction that all elements of one row (say, row 5) of $Minor_D$ are 0s. Now consider row 1–4 in $Minor_D$, *i.e.*, $Minor_D[1\text{–}4, *]$. Given (C2), each column contains exactly two 0s in row 1–4. With $2q-1$ columns, there are $2 \times (2q-1) = (4q-2)$ 0s. With 4 rows, there exists one row with at least $\lceil (4q-2)/4 \rceil = q$ 0s. As a result, this row and row 5 have equal elements in at least q columns, contradicting (C1).

LEMMA 3.6. *Under conditions (C1) and (C2), there are at least two columns of $Minor_D$ with non-overlapping* 1*s.*

Proof. Assume by way of contradiction that any two columns of $Minor_D$ have at least an overlapping 1. Suppose w.l.o.g. that

$$Minor_D[*, 1] = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

From Lemma 3.4, $Minor_D$ has a column that has a 0 in row 1. Since this column has an overlapping 1 with column 1, it must have a 1 in row 2. Assume w.l.o.g. that

$$Minor_D[*, 1\text{–}2] = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

From Lemma 3.4, $Minor_D$ has a column that has a 0 in row 2. Since this column has an overlapping 1 with both column 1 and column 2, it must have 1 in row 1 and 3. Assume w.l.o.g. that

$$Minor_D[*, 1\text{–}3] = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

But then every remaining column must have two 1s in row 1–3 (otherwise, its 1s will be non-overlapping with at least one of the first three columns). But then row 4 and 5 of $Minor_D$ contain no 1, contradicting Lemma 3.5.

LEMMA 3.7. *Under condition (C1), we can always construct a special indicator matrix M with $n \geq 1$ columns and all its non-zero rows form a $n \times n$ full-rank matrix on* $\mathbb{GF}(2)$.

Proof. We prove the lemma by performing induction on q. The base case ($q = 1$) is trivial, because in this case D contains only one column and every element of this column is unique (according to (C1)).

Now suppose Lemma 3.7 holds for D with $2q' - 1$ columns ($\forall q' < q$). Below we show that the lemma also holds for D with $2q-1$ columns. We assume that D has no column with a unique character (otherwise, M is trivial to construct). Then each column of $Minor_D$ either has five 0s, or has two 1s and three 0s.

Case 1. *At least one column (say, the last column) of $Minor_D$ has five* 0*s.* Now consider the first $2q-3$ columns. No two rows can have $q-1$ equal elements in the first $2q-3$ columns. Otherwise, when combined with the last column, we get two rows that share q elements, contradicting (C1). Notice that $2q-3 = 2 \times (q-1) - 1$. Hence condition (C1) holds on the first $2q-3$ columns of D. By our induction assumption, we can construct the desired M from these $2q-3$ columns.

Case 2. *No column of $Minor_D$ has five* 0*s.* In this case, every $Minor_D$ has exactly two 1s and three 0s. That is, condition (C2) holds. Given (C1) and (C2), we know from Lemma 3.6 that $Minor_D$ contains two columns with non-overlapping 1s. Assume w.l.o.g. that the first two columns of $Minor_D$ are:

$$Minor_D[*, 1\text{–}2] = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

From Lemma 3.5, there exists a column (say, column 3) of $Minor_D$ that has 1 in row 5, i.e., $Minor_D[5, 3] = 1$. Assume w.l.o.g. that $Minor_D[1, 3] = 1$ (the other cases are symmetric). That is,

$$Minor_D[*, 1\text{–}3] = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Thus, we have

$$D[*, 1\text{–}3] = \begin{bmatrix} a & c & e \\ a & c & f \\ b & d & f \\ b & d & f \\ b & c & e \end{bmatrix}.$$

Let $C = (1, 2, 3)$ and $W = (a, c, e)$. We then obtain

$$M = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

With $R = (1, 2, 5)$, the matrix $M[R, *]$ has rank 3 over $\mathbb{GF}(2)$, which is exactly what we need.

Given Lemma 3.7, the only remaining task in the proof for Theorem 3.2 is to show that condition (C1) holds on D. This turns out to be a direct consequence of (i) our choice of G and (ii) the fact that D is derived from distinct keys. Specifically, each row i of D can be computed as $D[i, *] = \vec{x_i}[I_q\ G]$, where I_q is a $q \times q$ identity matrix, and $[I_q\ G]$ is the horizontal concatenation of two matrices I_q and G. Since any square submatrix of G has full rank over prime field $\mathbb{Z}_p$, it is easy to show that any $q \times q$ submatrix of $[I_q\ G]$ has full rank over $\mathbb{Z}_p$. Therefore, from any q elements of $D[i, *]$, we can reconstruct the $\vec{x_i}$, and thereby the entire row $D[i, *] = \vec{x_i}[I_q\ G]$. As a result, if two rows of D share q elements, they are identical (because both rows can be reconstructed by the same q elements), which is impossible given the fact that D is derived from distinct keys. This completes our proof for Theorem 3.2.

3.3 Relaxed and efficient computation of $\vec{x}\,G$ on $\mathbb{Z}_p$.

Below we show how to compute $\vec{y} = \vec{x}\,G$ very efficiently on $\mathbb{Z}_p$ for general 5-universal hashing.

Multiplication through tabulation. Let $\vec{G}_i$. $i \in [q]$, be the q rows of the generator matrix G from Theorem 3.2. 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 5-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 3.2 is that there were $n \geq$ columns $c_0, \cdots, c_{n-1}$ and n special characters $w_0, \cdots, w_{n-1}$ ($w_j \in [p]$, $\forall j \in [n]$) such that (i) the resulting special indicator matrix M has exactly n non-zero rows $R = (r_0, \cdots, r_{n-1})$, and (ii) the $n \times n$ submatrix $M[R, *]$ has full rank over $\mathbb{GF}(2)$. By using regular integer addition rather than addition over $\mathbb{Z}_p$, a variable multiple of p is added to each occurrence of w_j, which effectively turns w_j into a set of special characters W_j. By splitting each column $M[*, j]$ into a set of columns indicating where each special character $w \in W_j$ appears in column $D[*, c_j]$, we obtain a new special indicator matrix M' with $n' \geq n$ columns and n non-zero rows. It is easy to see that with such splitting, M' still has rank n over $\mathbb{GF}(2)$. So we can take a subset of n columns of M' to form a full-rank square indicator matrix. As a result, our hash function remains 5-universal.

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. Thus, 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 $\bmod p$ 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 $a' = (a \wedge (2^c - 1)) + q - ((a \gg c) \wedge (2^c - 1))$, where $\gg$ 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 splits each special character into a set of special characters and does not reduce the rank of the resulting special indicator matrix. So our hash function remains 5-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 pre-computed constants, the compression is performed efficiently in one line of C code.

4 Experiments.

In this section, we experimentally evaluate (i) the speed of different hashing schemes, and (ii) the impact of different hash functions on linear probing. For (i) the basic target is to evaluate different 5-universal methods, but we also compare with the very fast multiplication-shift based methods to get a feel for the price paid for the 5-universality. For (ii) we perform the comparison within the context of linear probing. Here the multiplication-shift based methods represent the natural choice of a practical hash function for someone not aware that a higher degree of universality is needed, and our goal is to see how our theory-founded choice of a 5-universal hash function performs against this more naïve practical choice.

4.1 Hashing. We first compare the speed of different hashing schemes.

Experiments. We have implemented our schemes and *CW-trick* in C. Here both schemes are understood to be 5-universal, so our character tables are 5-universal, and for CW-trick we use a degree 4 polynomial in (1.2). To evaluate their performance, we record the total running time for performing 10 million hash computations. In [15], we simply use $1, 2, ..., 10^7$ as the sequence of input keys. But our results suggest that such an input sequence can give tabulation-based hashing an unfair advantage. Specifically, since the key is incremented by 1 each time, the higher order bits of the key change only infrequently. As a result, the results of table lookups involving higher-order characters often reside in the cache already, thus reducing the need for memory access in many cases. For a more fair comparison, we generate a million distinct random input keys 10-universally and stored them in an array. We then cycle through the array of random keys 10 times, resulting in a total of 10 million hash computations.

Findings. Table 1 compares the different algorithms in terms of average hashing overhead (in nanoseconds) on two computers with different architectures. For $w = 32, 48, 64$, the goal is to produce w-bit hash values from w-bit keys. Univ and Univ2 are the very fast multiplication-shift based methods from §1.1.1 for plain universal hashing and 2-universal hashing, respectively. The actual code for Univ and Univ2 can be found in § A.2. CWtrick32, CWtrick48 and CWtrick64 are CW-trick schemes as described in §1.1.1 with Mersenne primes $2^{61} - 1$, $2^{61} - 1$, and $2^{89} - 1$, respectively. The actual code for CWtrick32, CWtrick48, and CWtrick64 can be found in § A.9, § A.10, and § A.11, respectively. All the codes are fairly tuned. ShortTable and CharTable are instances of our new tabulation based 5-universal hashing schemes from §3 with 16-bit and 8-bit input characters, respectively. To minimize their memory requirement, we enable compression (described in § 3.3) in all our experiments. The code for ShortTable and CharTable with different key lengths can be found in § A.3–A.8.

		hashing time (nanoseconds)	
bits	algorithm	computer A	computer B
32	Univ	1.87	3.07
32	Univ2	5.79	3.22
32	CWtrick32	99.83	22.94
32	ShortTable32	17.06	21.79
32	CharTable32	**10.18**	**12.70**
48	CWtrick48	139.24	40.34
48	ShortTable48	217.36	193.74
48	CharTable48	**50.75**	**17.37**
64	CWtrick64	324.48	59.08
64	ShortTable64	278.33	235.27
64	CharTable64	**75.99**	**22.12**

Table 1: Average time per hash computation for 10 million hash computations on computer A (single-core Intel Xeon 3.2 GHz processor with 32-bit address), and B (dual-core AMD Opteron 2 GHz processor with 64-bit address). For each key length, we highlight the best performance on each computer.

It is interesting to compare the performance of Computer A and B since A is a 32-bit architecture while B is 64-bit. This shows up nicely in the difference between Univ and Univ2. The difference between the schemes is that Univ does a 32-bit multiplication where Univ2 does a 64-bit multiplication. On Computer A we have that Univ is almost three times faster than Univ2. This could indicate that it uses its fast 32-bit multiplication to simulate 64-bit multiplication. On Computer B the difference in speed is less than 10%, so this 64-bit architecture gains very little from working on 32-bit numbers. The difference between the two computers shows up even more strongly in the CWtrick implementations which are all dominated by 64-bit multiplications. Here Computer B appears to be 3-5 times faster than Computer A. Note that because of differences in compilation and pipelining, we can never hope to give exact prediction of performance based the speed of individual operations. Another issue is that the timing results include the time spent on reading the keys sequentially from an array. This adds at most a nanosecond to most timing results. We find no simple way of correctly subtracting the effect, *e.g.*, sometimes adding more operations reduced running times due to differences in the compilation, and even the cycle counter isn't reliable in measuring the cost of the hash computation itself. Nevertheless it is clear that Computer B is much better at the 64-bit operations needed for CWtrick.

If we now turn our attention to CharTable32 which is dominated by memory look-ups from small tables, each with around 2^8 entries, we see that the two computers have a similar performance with Computer A being slightly faster. However, CharTable48 and CharTable64 becomes relatively slower on Computer A. The number of table look-ups with CharTable is $2(w/8) - 1$, hence 7, 11, and 15 look-ups for $w = 32, 48, 64$. This increase seems reasonably matched on Computer B but on Computer A, we have a marked jump going from 32 to 48 bits. The likely explanation is that we start working more with 64-bit integers, which is comparatively less efficient on Computer A.

Considering ShortTable, we see that it is always worse than CharTable. The code is simper for ShortTable, but the individual tables are much larger, with 2^{16} instead of 2^8 entries. With ShortTable32 we use 3 such tables with 32-bit values, adding up to less than 1MB, but With ShortTable48, we use 5 tables with 64-bit values, adding up to more than 2.5 MB, and then the memory performance starts slowing down. For comparison, CharTable64 uses 15 tables, each with 2^8 64-bit integers, so the total tabulation space is only about 15K, which fits easily in cache.

Across the experiments, for 5-universal hashing, we see that CharTable consistently gives the best performance regardless of the key length and the computer architecture. CharTable consistently outperforms ShortTable, especially for longer keys. Compared with CWtrick (which is the previous fastest method for 5-universal hashing), CharTable gains a factor of 2 to 10 in speed. The advantage is much smaller on Computer B whose fast 64-bit arithmetic is a big win for CWtrick, yet the advantage is always more than a factor of 1.8.

4.2 Linear probing. Linear probing is one of the most popular implementations of dynamic hash tables storing all keys in a single array. Below we generally assume that the keys fill half the array. When we get a key, we first hash it to a location. Next we probe consecutive locations until the key or an empty location is found. Giving birth to the analysis of algorithm, Knuth [7] proved that linear probing uses an expected constant number of probes if the hash function is truly random. More recently, Pagh *et al.* [11] proved that 5-universal hashing suffices for an expected constant number of probes per update or search, and this is for any possible set of input keys. We also note that Thorup [14] has shown that we preserve this constant expected time if we first do plain universal hashing into a domain of the same size n as the number of keys. Thus if we can provide a fast 5-universal hashing for 32- or at most 64-bit keys, then we get a fast implementation of linear probing with provably good expected performance.

Our next question is then if our 5-universal hashing with its good theoretical performance would also perform better in practice on some simple to understand realistic data. We know that these data should not be random, for then any reasonable hash function would work well. In fact, in [9] it is shown that plain universal hashing suffices as long as there is enough entropy in the data. As our competing practical hash function, we use the very fast multiplication-shift based methods Univ and Univ2. These methods represent the natural choice of a practical hash function for someone not aware that a higher degree of universality is needed, and our goal is to see how our theory-founded choice of a 5-universal hash function performs against this more naïve practical choice. We note that this choice is not that naïve, for plain universal hashing suffices for other implementations of hash tables like simple chaining, and it is only recently [11] that we have learned that the theoretical performance of linear probing is so sensitive to the degree of universality.

It was already known from [5] that structured input can cause linear probing to be slower than other methods, but in [5] the slowness was largely due to the use of the old-fashioned direct 2-universal method from (1.2). This particular hash function was proven to be bad also for linear probing in [11]. Here we compete against the multiplication-shift based methods that are an order of magnitude faster. Also our experiments are special in that that we do many experiments to study robustness.

Experiments. We conduct experiments to evaluate the impact of hash functions on the performance of linear probing. We construct a hash table with 2^{21} entries. We then insert entries into the hash table one by one until the array is half full. From then on we perform 10 million insertion/deletion cycles. During each cycle, we first insert a new key into the hash table, and then delete an old key from the hash table, which was inserted into the hash table a million steps back. We then compute the average amount of time spent for each update operation (i.e., either insertion or deletion). In addition to such timing results, we report the average number of linear probes involved per update, which is independent of machine configurations.

For our experiments, we construct two very different key sequences:

- *A dense interval.* In this case, we use a random permutation of $[2^{20}]$ as the key sequence. To do so, we first generate 2^{20} distinct 32-bit random numbers 10-universally and then sort these numbers to obtain a permutation of their original indices. At any point, we have the last 1 million keys in the table, representing roughly 95% of $[2^{20}]$.

- *Random keys.* In this case, we generate 2^{20} distinct 32-

bit random numbers 10-universally and use them as the key sequence. This is also the sequence used when we test the speed of the hash functions above.

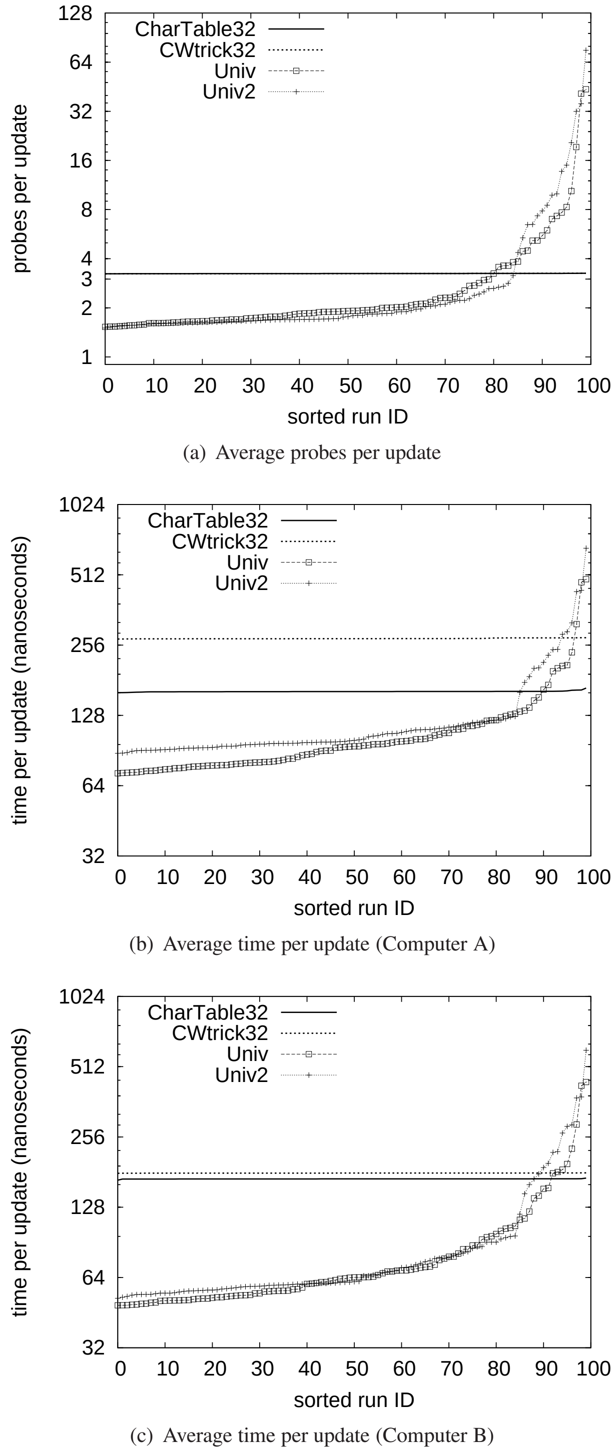
We store the constructed input key sequence in an array (of size 2^{20}) and then cycle through the array to obtain the next key to insert or delete. For the same input key sequence, we repeat the experiment 100 times. In each experiment, we obtain a different set of random numbers (from `random.org`) and use them to initialize all the hash functions.

Findings. Figure 1–2 show the results. In each figure we have a particular input key sequence for which we study linear probing with different hash functions. For each hash function, we consider 100 different sets of random seeds, and plot the performance from best to worst. In (a) we have the average number of probes per operation over 10 million insert/delete cycles; in (b) and (c) we have the average time per operation on Computer A and B, respectively. The random seeds are the same across (a)–(c), so the same probe count holds for both computers.

We first consider the combinatorial probe count measure (a). In Figure 1 we have a dense interval. For such highly structured input, neither Univ nor Univ2 is robust. For some experiments, the use of Univ and Univ2 require a significant number of linear probes per insertion/deletion. In contrast, with our 5-universal schemes, CharTable32 and CWtrick32, we do not see any obvious difference between the best and the worst experiment. The average probe count ranges from 3.23 to 3.26. The much smaller variance for 5-universal hashing is expected, because the result from [11] also limits the variance on the probes per operation. More precisely, to bound the expected number of probes, they show in the proof of [10, Theorem 4.3] that the probability of doing more than ℓ probes is $O(1/\ell^2)$, hence that the expected number is $O(\sum_{\ell\in[n]} 1/\ell^2) = O(1)$. We can also use this to bound the variance, which within a factor 2 can be computed by letting the ℓth probe pay ℓ, leading to a variance bound of $O(\sum_{\ell\in[n]} \ell/\ell^2) = O(\log n)$. Overall, for 10–20% experiments (which use different random seeds for initializing the hash function), CharTable32 and CWtrick32 significantly outperform Univ and Univ2 in terms of the average probe count.

Now consider instead the average probe count in Figure 2 (a) for a random set of keys. These are 10-universal, so in fact, much more random than our 5-universal hash functions CharTable32 and CWtrick32. With so much randomness in the keys, the limited randomness in the hash functions has no impact, and now we see that all schemes have a robust average probe count of around 3.24. In particular, the heavy tails disappear from Univ and Univ2.

We now switch our attention to the average time spent per update on the two computers. In essence this is the cost of hash computation plus the cost of memory access for the

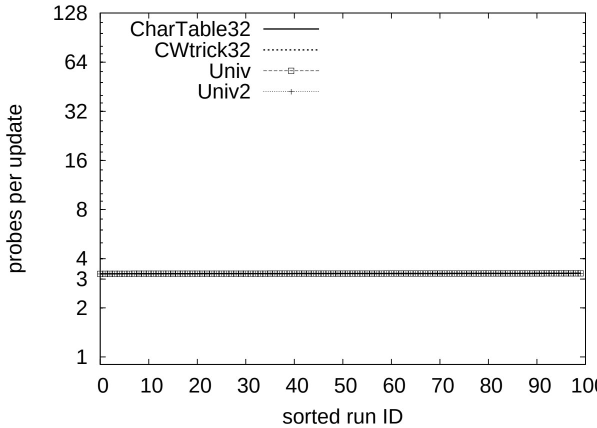


(a) Average probes per update

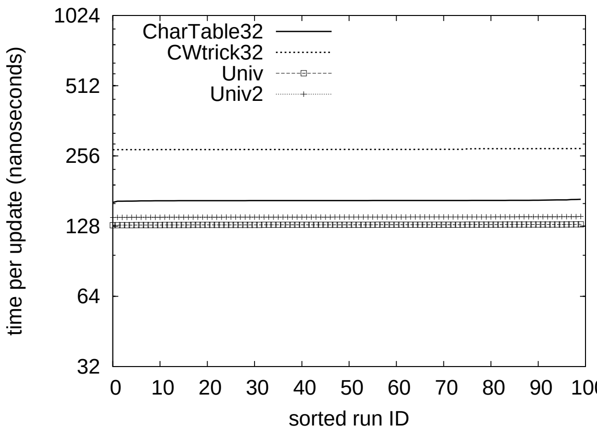
(b) Average time per update (Computer A)

(c) Average time per update (Computer B)

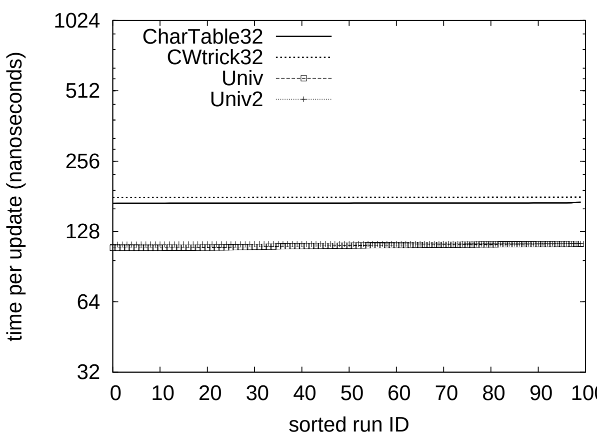
Figure 1: Impact of hash functions on linear probing performance with permuted input sequence. Each data point represents the result for one experiment run.



(a) Average probes per update



(b) Average time per update (Computer A)



(c) Average time per update (Computer B)

Figure 2: Impact of hash functions on linear probing performance with random input sequence. Each data point represents the result for one experiment run.

probes in the table. The latter is dominated by the cache miss from the initial probe at the hash location. For the random keys in Figure 2, we essentially have the same number of probes with all the hash function, so the differences in time are due to the differences in hash computation. Thus it is not surprising that we see the same ordering as the one found for the hash computations alone in Table 1.

In Figure 2 (b) we have the results for Computer A. A slightly surprising thing is that the differences are bigger than those in Table 1. More precisely, in Figure 2 (b) we see that CharTable32 is about 25 nanoseconds slower than Univ, and CWtrick32 is about 130 nanoseconds slower. In both cases, this is much more than the cost of computing the hash function itself. It appears that the optimizing compiler does not do as good a job when faced with the more complicated hash functions.

In Figure 2 (c) we have the results for Computer B. Again we see CharTable32 and CWtrick32 are slower than what we would expect from Table 1, and relatively speaking, the difference between CharTable32 and CWtrick32 is reduced, yet we still have CharTable32 coming out as the fastest 5-universal scheme.

If we now consider Figure 1 (b) and (c), we see the combined effect of differences in hashing speed and differences in number of probes. This gives Univ and Univ2 an additive advantage compared with the pure probe count in (a), so now it is only for 10% of the cases that CharTable32 and CWtrick32 perform better than Univ and Univ2. Yet the heavy tails persist, so Univ and Univ2 are still much less robust.

4.3 Summary. Among 5-universal schemes, we have seen that our tabulation based scheme CharTable is much faster than the classic CWtrick. When the hashing is studied in isolation (c.f. Table 1) the difference is by a factor 2 to 10. The smallest gain is on Computer B which has really fast 64-bit multiplications as needed by CWtrick, but seemingly slightly slower memory. When used inside linear probing (c.f. Figures 1–2 (b)–(c)), we see that CharTable continues to outperform CWtrick. We therefore recommend CharTable as the fastest choice for a 5-universal hash function for computers with a reasonably fast cache.

Our other research question is how the 5-universal hashing with its proven theoretical performance would perform against the fast practical multiplication-shift based choices Univ and Univ2. On the random data from Figure 2, we see that we lose about 10% in speed on Computer A, and about 40% on computer B. On the other hand, Univ and Univ2 have a very heavy tail on the dense interval in Figure 1. Thus we recommend using CharTable over Univ and Univ2 if robustness is an issue and we have no guarantee that the input is random.

References

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

[2] 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.

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

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

[5] G. L. Heileman and W. Luo. How caching affects hashing. In *Proc. 7th ALENEX*, pages 141–154, 2005.

[6] H. Karloff and P. Raghavan. Randomized algorithms and pseudorandom numbers. *J. ACM*, 40(3):454–476, 1993.

[7] D. E. Knuth. Notes on "open" addressing, 1963. Unpublished memorandum. Available at citeseer.ist.psu.edu/knuth63notes.html.

[8] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973. ISBN 0-201-03803-X.

[9] M. Mitzenmacher and S. P. Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *Proc. 19th SODA*, pages 746–755, 2008.

[10] A. Pagh, R. Pagh, and M. Ruzic. Linear probing with constant independence. In *Proc. 39th STOC*, pages 318–327, 2007.

[11] A. Pagh, R. Pagh, and M. Ruzic. Linear probing with constant independence. *SIAM J. Computing*, 39(3):1107–1120, 2009.

[12] A. Siegel. On universal classes of extremely random constant-time hash functions. *SIAM J. Comput.*, 33(3):505–543, 2004.

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

[14] M. Thorup. String hashing for linear probing. In *Proc. 20th SODA*, pages 655–664, 2009.

[15] M. Thorup and Y. Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proc. 15th SODA*, pages 608–617, 2004.

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

A Code.

A.1 Common data types and macros.

```
typedef unsigned char       INT8;
typedef unsigned short      INT16;
typedef unsigned int        INT32;
typedef unsigned long long  INT64;
typedef INT64               INT96[3];

// different views of a 64-bit double word
typedef union {
  INT64 as_int64;
  INT16 as_int16s[4];
} int64views;

// different views of a 32-bit single word
typedef union {
  INT64 as_int32;
  INT16 as_int16s[2];
  INT8  as_int8s[4];
} int32views;

typedef struct {
  INT64 h;
  INT64 u;
  INT32 v;
} Entry;

// extract lower and upper 32 bits from INT64
const INT64 LowOnes = (((INT64)1)<<32)-1;
#define LOW(x)  ((x)&LowOnes)
#define HIGH(x) ((x)>>32)
```

A.2 Multiplication-shift based hashing for 32-bit keys.

```
/* plain universal hashing for 32-bit key x
   A is a random 32-bit odd number */
inline INT32 Univ(INT32 x, INT32 A) {
  return (A*x);
}

/* 2-universal hashing for 32-bit key x
   A and B are random 64-bit numbers */
inline INT32 Univ2(INT32 x,
                   INT64 A, INT64 B)
{
  return (INT32) ((A*x + B) >> 32);
}
```

A.3 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 pre-computed tables */
inline INT32 ShortTable32(INT32 x,
  INT32 T0[], INT32 T1[], INT32 T2[])
{
  INT32 x0, x1, x2;
  x0 = x&65535;
  x1 = x>>16;
  x2 = x0 + x1;
  x2 = compressShort32(x2);  // optional
  return T0[x0] ^ T1[x1] ^ T2[x2];
}

// optional compression
inline INT32 compressShort32(INT32 i) {
  return 2 - (i>>16) + (i&65535);
}
```

A.4 Tabulation based hashing for 32-bit keys using 8-bit characters.

```
/* tabulation based hashing for 32-bit key x
   using 8-bit characters.
```

```
   T0, T1 ... T6 are pre-computed tables */
inline INT32 CharTable32(int32views x,
  INT32 *T0[], INT32 *T1[],
  INT32 *T2[], INT32 *T3[],
  INT32 T4[], INT32 T5[], INT32 T6[])
{
  INT32 *a0, *a1, *a2, *a3, c;

  a0 = T0[x.as_int8s[0]];
  a1 = T1[x.as_int8s[1]];
  a2 = T2[x.as_int8s[2]];
  a3 = T3[x.as_int8s[3]];

  c  = a0[1] + a1[1] + a2[1] + a3[1];
  c  = compressChar32(c);  // optional

  return
    a0[0] ^ a1[0] ^ a2[0] ^ a3[0] ^
    T4[c&1023] ^ T5[(c>>10)&1023] ^
    T6[c>>20];
}

// optional compression
inline INT32 compressChar32(INT32 i) {
  const INT32 Mask1 =
    (((INT32)3)<<20) + (((INT32)3)<<10) + 3;
  const INT32 Mask2 =
    (((INT32)255)<<20) +
    (((INT32)255)<<10) + 255;
  const INT32 Mask3 =
    (((INT32)3)<<20) + (((INT32)3)<<10) + 3;

  return Mask1 + (i&Mask2) - ((i>>8)&Mask3);
}
```

A.5 Tabulation based hashing for 48-bit keys using 16-bit characters.

```
/* tabulation based hashing for 48-bit key x
   using 16-bit characters.
   T0, T1 ... T4 are pre-computed tables */
inline INT64 ShortTable48(int64views x,
  INT64 *T0[], INT64 *T1[], INT64 *T2[],
  INT64 T3[], INT64 T4[])
{
  INT64 *a0, *a1, *a2, c;

  a0 = T0[x.as_int16s[0]];
  a1 = T1[x.as_int16s[1]];
  a2 = T2[x.as_int16s[2]];

  c = a0[1] + a1[1] + a2[1];
  c = compressShort48(c);   // optional

  return
    a0[0] ^ a1[0] ^ a2[0] ^
    T3[c&262143] ^ T4[c>>18];
}

// optional compression
inline INT32 compressShort48(INT64 i) {
  const INT64 Mask1 = (((INT64)3)<<18) + 3;
  const INT64 Mask2 =
    (((INT64)65535)<<18) + 65535;
  const INT64 Mask3 = (((INT64)3)<<18) + 3;

  return Mask1 + (i&Mask2) - ((i>>16)&Mask3);
}
```

A.6 Tabulation based hashing for 48-bit keys using 8-bit characters.

```
/* tabulation based hashing for 48-bit key x
   using 8-bit characters.
   T0, T1 ... T10 are pre-computed tables */
inline INT64 CharTable48(int64views x,
  INT64 *T0[], INT64 *T1[], INT64 *T2[],
  INT64 *T3[], INT64 *T4[], INT64 *T5[],
  INT64  T6[], INT64 T7[], INT64 T8[],
  INT64 T9[], INT64 T10[])
{
  INT64 *a0, *a1, *a2, c;

  a0 = T0[x.as_int8s[0]];
  a1 = T1[x.as_int8s[1]];
  a2 = T2[x.as_int8s[2]];
  a3 = T2[x.as_int8s[3]];
  a4 = T2[x.as_int8s[4]];
  a5 = T2[x.as_int8s[5]];

  c = a0[1] + a1[1] + a2[1] +
      a3[1] + a4[1] + a5[1];
  c = compressChar48(c);   // optional

  return
    a0[0] ^ a1[0] ^ a2[0] ^
    a3[0] ^ a4[0] ^ a5[0] ^
    T6[(c&2047)] ^ T7[((c>>11)&2047)] ^
    T8[((c>>22)&2047)] ^
    T9[((c>>33)&2047)] ^ T10[c>>44];
}

// optional compression
inline INT32 compressChar48(INT64 i) {
  const INT64 Mask1 = 5 +
    (((INT64)5)<<11) + (((INT64)5)<<22) +
    (((INT64)5)<<33) + (((INT64)5)<<44);
  const INT64 Mask2 = 255 +
    (((INT64)255)<<11) + (((INT64)255)<<22) +
    (((INT64)255)<<33) + (((INT64)255)<<44);
  const INT64 Mask3 = 7 +
    (((INT64)7)<<11) + (((INT64)7)<<22) +
    (((INT64)7)<<33) + (((INT64)7)<<44);

  return Mask1 + (i&Mask2) - ((i>>8)&Mask3);
}
```

A.7 Tabulation based hashing for 64-bit keys using 16-bit characters.

```
/* tabulation based hashing for 64-bit key x
   using 16-bit characters.
   T0, T1 ...  T6 are pre-computed tables */
inline INT64 ShortTable64(int64views x,
```

```
  INT64 *T0[], INT64 *T1[],
  INT64 *T2[], INT64 *T3[],
  INT64 T4[], INT64 T5[], INT64 T6[])
{
  INT64 *a0, *a1, *a2, *a3, c;

  a0 = T0[x.as_int16s[0]];
  a1 = T1[x.as_int16s[1]];
  a2 = T2[x.as_int16s[2]];
  a3 = T3[x.as_int16s[3]];

  c = a0[1] + a1[1] + a2[1] + a3[1];
  c = compressShort64(c);   // optional

  return
    a0[0] ^ a1[0] ^ a2[0] ^ a3[0] ^
    T4[c&2097151] ^
    T5[(c>>21)&2097151] ^ T6[c>>42];
}

// optional compression
inline INT64 compressShort64(INT64 i) {
  const INT64 Mask1 = 4 +
    (((INT64)4)<<21) + (((INT64)4)<<42);
  const INT64 Mask2 = 65535 +
    (((INT64)65535)<<21) +
    (((INT64)65535)<<42);
  const INT64 Mask3 = 31 +
    (((INT64)31)<<21) + (((INT64)31)<<42);

  return Mask1 + (i&Mask2) - ((i>>16)&Mask3);
}
```

A.8 Tabulation based hashing for 64-bit keys using 8-bit characters.

```
/* tabulation based hashing for 64-bit key x
   using 8-bit characters.
   T0, T1... T14 are pre-computed tables */
inline INT64 CharTable64(int64views x,
  Entry T0[], Entry T1[], Entry T2[],
  Entry T3[], Entry T4[], Entry T5[],
  Entry T6[], Entry T7[], INT64 T8[],
  INT64 T9[], INT64 T10[], INT64 T11[],
  INT64 T12[], INT64 T13[], INT64 T14[])
{
  Entry *a0, *a1, *a2, *a3,
        *a4, *a5, *a6, *a7;
  INT64 c0;
  INT32 c1;

  a0 = &T0[x.as_int16s[0]];
  a1 = &T1[x.as_int16s[1]];
  a2 = &T2[x.as_int16s[2]];
  a3 = &T3[x.as_int16s[3]];
  a4 = &T4[x.as_int16s[4]];
  a5 = &T5[x.as_int16s[5]];
  a6 = &T6[x.as_int16s[6]];
  a7 = &T7[x.as_int16s[7]];

  c0  = a0->u + a1->u + a2->u + a3->u +
        a4->u + a5->u + a6->u + a7->u;
  c1  = a0->v + a1->v + a2->v + a3->v +
        a4->v + a5->v + a6->v + a7->v;

  c0 = compressChar64_0(c0);   // optional
  c1 = compressChar64_1(c1);   // optional

  return
    a0->h ^ a1->h ^ a2->h ^ a3->h ^
    a4->h ^ a5->h ^ a6->h ^ a7->h ^
    T8[(c0&2043)] ^ T9[((c0>>11)&2043)] ^
    T10[((c0>>22)&2043)] ^
    T11[((c0>>33)&2043)] ^ T12[(c0>>44)] ^
    T13[(c1&2043)] ^ T14[(c1>>11)];
}

// optional compression
inline INT64 compressChar64_0(INT64 i) {
  const INT64 Mask1 = 7 +
    (((INT64)7)<<11) + (((INT64)7)<<22) +
    (((INT64)7)<<33) + (((INT64)7)<<44);
  const INT64 Mask2 = 255 +
    (((INT64)255)<<11) + (((INT64)255)<<22) +
    (((INT64)255)<<33) + (((INT64)255)<<44);
  const INT64 Mask3 = 7 +
    (((INT64)7)<<11) + (((INT64)7)<<22) +
    (((INT64)7)<<33) + (((INT64)7)<<44);

  return Mask1 + (i&Mask2) - ((i>>8)&Mask3);
}

// optional compression
inline INT32 compressChar64_1(INT32 i) {
  const INT64 Mask1 = (((INT64)7)<<11) + 7;
  const INT64 Mask2 =
    (((INT64)255)<<11) + 255;
  const INT64 Mask3 = (((INT64)7)<<11) + 7;

  return Mask1 + (i&Mask2) - ((i>>8)&Mask3);
}
```

A.9 CW trick for 32-bit keys with prime $2^{61} - 1$.

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

/* Computes ax+b mod Prime,
   possibly plus 2*Prime,
   exploiting the structure of Prime. */
inline INT64 MultAddPrime32(INT32 x,
  INT64 a, INT64 b)
{
  INT64 a0,a1,c0,c1,c;
  a0 = LOW(a)*x;
  a1 = HIGH(a)*x;
  c0 = a0+(a1<<32);
  c1 = (a0>>32)+a1;
  c  = (c0&Prime)+(c1>>29)+b;
  return c;
}

// CWtrick for 32-bit key x (Prime = 2^61-1)
inline INT64 CWtrick32(INT32 x, INT64 A,
  INT64 B, INT64 C, INT64 D, INT64 E)
```

```
{
  INT64 h;
  h = MultAddPrime32(
      MultAddPrime32(
      MultAddPrime32(
      MultAddPrime32(x,A,B),x,C),x,D),x,E);
  h = (h&Prime)+(h>>61);
  if (h>=Prime) h-=Prime;
  return h;
}
```

A.10 CW trick for 48-bit keys with prime $2^{61}-1$.

```
/* Computes ax+b mod Prime,
   possibly plus 2*Prime,
   exploiting the structure of Prime. */
inline INT64 MultAddPrime(INT64 x,
  INT64 a, INT64 b)
{
  INT64 x0, x1, c0, c1, c,
        a0, a1, ax00, ax01_10, ax11;

  x0 = LOW(x); x1 = HIGH(x);
  a0 = LOW(a); a1 = HIGH(a);

  ax00    = a0*x0;
  ax11    = a1*x1;
  ax01_10 = a0*x1 + a1*x0;

  c0 = ax00 + (ax01_10<<32);
  c1 = (ax00>>61) + (ax01_10>>29) +
       (ax11<<3);
  c  = (c0&Prime61) + c1 + b;
  c  = (c&Prime61) + (c>>61);

  return c;
}

// CWtrick for 48-bit key x (Prime = 2^61-1)
inline INT64 CWtrick48(INT32 x, INT64 A,
  INT64 B, INT64 C, INT64 D, INT64 E)
{
  INT64 h;
  h = MultAddPrime(
      MultAddPrime(
      MultAddPrime(
      MultAddPrime(x,A,B),x,C),x,D),x,E);
  h = (h&Prime)+(h>>61);
  if (h>=Prime) h-=Prime;
  return h;
}
```

A.11 CW trick for 64-bit keys using prime $2^{89}-1$.

```
const INT64 Prime89_0  = (((INT64)1)<<32)-1;
const INT64 Prime89_1  = (((INT64)1)<<32)-1;
const INT64 Prime89_2  = (((INT64)1)<<25)-1;
const INT64 Prime89_21 = (((INT64)1)<<57)-1;

/* Computes (r mod Prime89) mod 2^64,
   exploiting the structure of Prime89 */
inline INT64 Mod64Prime89(INT96 r) {
  INT64 r0, r1, r2;

  // r2r1r0 = r&Prime89 + r>>89
  r2 = r[2];
  r1 = r[1];
  r0 = r[0] + (r2>>25);
  r2 &= Prime89_2;

  return (r2 == Prime89_2 &&
          r1 == Prime89_1 &&
          r0 >= Prime89_0) ?
         (r0 - Prime89_0) : (r0 + (r1<<32));
}

/* Computes a 96-bit r such that
   r mod Prime89 == (ax+b) mod Prime89
   exploiting the structure of Prime89. */
inline void MultAddPrime89(INT96 r, INT64 x,
  INT96 a, INT96 b)
{
  INT64 x1, x0, c21, c20, c11, c10, c01, c00;
  INT64 d0, d1, d2, d3;
  INT64 s0, s1, carry;

  x1 = HIGH(x);  x0 = LOW(x);

  c21 = a[2]*x1;  c20 = a[2]*x0;
  c11 = a[1]*x1;  c10 = a[1]*x0;
  c01 = a[0]*x1;  c00 = a[0]*x0;

  d0 = (c20>>25)+(c11>>25)+
       (c10>>57)+(c01>>57);
  d1 = (c21<<7);
  d2 = (c10&Prime89_21) + (c01&Prime89_21);
  d3 = (c20&Prime89_2) +
       (c11&Prime89_2) + (c21>>57);

  s0 = b[0] + LOW(c00) + LOW(d0) + LOW(d1);
  r[0] = LOW(s0);  carry = HIGH(s0);

  s1 = b[1] + HIGH(c00) + HIGH(d0) +
       HIGH(d1) + LOW(d2) + carry;
  r[1] = LOW(s1);
  carry = HIGH(s1);

  r[2] = b[2] + HIGH(d2) + d3 + carry;
}

// CWtrick for 64-bit key x (Prime = 2^89-1)
inline INT64 CWtrick64(INT64 x, INT96 A,
  INT96 B, INT96 C, INT96 D, INT96 E)
{
  INT96 r;
  MultAddPrime89(r,x,A,B);
  MultAddPrime89(r,x,r,C);
  MultAddPrime89(r,x,r,D);
  MultAddPrime89(r,x,r,E);
  return Mod64Prime89(r);
}
```