Strongly universal string hashing is fast

DANIEL LEMIRE[1][*] AND OWEN KASER[2]

[1]LICEF Research Center, TELUQ, Université du Québec, Canada
[2]Department of CSAS, University of New Brunswick, Canada
Email: lemire@gmail.com

We present fast strongly universal string hashing families: they can process data at a rate of 0.2 CPU cycle per byte. Maybe surprisingly, we find that these families—though they require a large buffer of random numbers—are often faster than popular hash functions with weaker theoretical guarantees. Moreover, conventional wisdom is that hash functions with fewer multiplications are faster. Yet we find that they may fail to be faster due to operation pipelining. We present experimental results on several processors including low-power processors. Our tests include hash functions designed for processors with the Carry-Less Multiplication (CLMUL) instruction set. We also prove, using accessible proofs, the strong universality of our families.

1. INTRODUCTION

For 32-bit integers, random hashing with good theoretical guarantees can be just as fast as popular alternatives [1]. In turn, these guarantees ensure the reliability of various algorithms and data structures: set intersection [2], frequent-item mining [3], count estimation [4, 5], and hash tables [6, 7, 8]. We want to show that we can also get good theoretical guarantees over larger objects (such as strings) without sacrificing speed. For example, we consider variable-length strings made of 32-bit characters: all data structures can be represented as such strings, up to some padding.

We restrict our attention to hash functions mapping strings to L-bit integers, that is, integers in $[0, 2^L)$ for some positive integer L. In random hashing, we select a hash function at random from a family [9, 10]. The hash function can be chosen whenever the software is initialized. While random hashing is not yet commonplace, it can have significant security benefits [11] in a hash table: without randomness, an attacker can more easily exploit the fact that adding n keys hashing to the same value typically takes quadratic time ($\Theta(n^2)$). For this reason, random hashing was adopted in the Ruby language as of version 1.9 [12] and in the Perl language as of version 5.8.1.

A family of hash functions is k-wise independent (or k-independent) if the hash values of any k distinct elements are independent. For example, a family is pairwise independent—or strongly universal—if given any two distinct elements s and s', their hash values $h(s)$ and $h(s')$ are independent:

$$P(h(s) = y | h(s') = y') = P(h(s) = y)$$

for any two hash values y, y'. (Some authors prefer the terms 2-independent or 2-universal to describe strongly universal hash families.) When a hashing family is not strongly universal, it can still be universal if the probability of a collision is no larger than if it were strongly universal: $P(h(s) = h(s')) \leq 1/2^L$ when 2^L is the number of hash values. If the collision probability is merely bounded by some ϵ larger than $1/2^L$ but smaller than 1 ($P(h(s) = h(s')) \leq \epsilon < 1$), we have an almost universal family. However, strong universality might be more desirable than universality or almost universality:

- We say that a family is uniform if all hash values are equiprobable ($P(h(s) = y) = 1/2^L$ for all y and s): strongly universal families are uniform, but universal or almost universal families may fail to be uniform. To see that universality fails to imply uniformity, consider the family made of the two functions over 1-bit integers (0,1): the identity and a function mapping all values to zero. The probability of a collision between two distinct values is exactly $1/2$ which ensures universality even though we do not have uniformity since $P(h(0) = 0) = 1$.

- Moreover, if we have strong universality over L bits, then we also have it over any subset of bits. The corresponding result may fail for universal and almost universal families: we might have universality over L bits, but fail to have almost universality over some subset of bits. Consider the non-uniform but universal family $\{h(x) = x\}$ over L-bit integers: if we keep only the least significant L' bits $(0 < L' < L)$, universality is lost since $h(0) \bmod 2^{L'} = h(2^{L'}) \bmod 2^{L'}$.

There is no need to use slow operations such as modulo operations, divisions or operations in finite fields to have strong universality. In fact, for short strings having few distinct characters, Zobrist hashing requires nothing more than table look-ups and bitwise exclusive-or operations, and it is more than strongly universal (3-wise independent) [13, 14]. Unfortunately, it becomes prohibitive for long strings as it requires the storage of nc random numbers where n is the maximal length of a string and c is the number of distinct characters.

A more practical approach to strong universality is Multilinear hashing (§ 2). Unfortunately, it normally requires that the computations be executed in a finite field. Some processors have instructions for finite fields (§ 4) or they can be emulated with a software library (§ 5.3). However, if we are willing to double the number of random bits, we can implement it using regular integer arithmetic. Indeed, using an idea from Dietzfelbinger [15], we implement it using only one multiplication and one addition per character (§ 3). We further attempt to speed it up by reducing the number of multiplications by half. We believe that these families are the fastest strongly universal hashing families on current computers. We evaluate these hash families experimentally (§ 5):

- Using fewer multiplications has often improved performance, especially on low-power processors [16]. Yet trading away the number of multiplications fails to improve (and may even degrade) performance on several processors according to our experiments—which include low-power processors. However, reducing the number of multiplications is beneficial on some processors (e.g., AMD V120).
- We also find that strongly universal hashing may be computationally inexpensive compared to common hashing functions, as long as we ignore the overhead of generating long strings of random numbers. In effect—if memory is abundant compared to the length of the strings—the strongly universal Multilinear family is faster than many of the commonly used alternatives.
- We consider hash functions designed for hardware supported carry-less multiplications (§ 4). This support should drastically improve the speed of some operations over binary finite fields ($GF(2^L)$). Unfortunately, we find that the carry-less hash

functions fail to be competitive (§ 5.4).

2. THE MULTILINEAR FAMILY

The Multilinear hash family is one of the simplest strongly universal families [9]. It takes the form of a scalar product between random values (sometimes called keys) and string components, where operations are over a finite field:

$$h(s) = m_1 + \sum_{i=1}^{n} m_{i+1} s_i.$$

The hash function h is defined by the randomly generated values $m_1, m_2, \ldots$ It is strongly universal over fixed-length strings. We can also apply it to variable-length strings as long as we forbid strings ending with zero. To ensure that strings never end with zero, we can append a character value of one to all variable-length strings.

An apparent limitation of this approach is that strings cannot exceed the number of random values. In effect, to hash 32-bit strings of length n, we need to generate and store $32(n + 1)$ random bits using a finite field of cardinality 2^{32}. However, Stinson [17] showed that strong universality requires at least $1 + a(b-1)$ hash functions where a is the number of strings and b is the number of hash values. Thus, if we have 32-bit strings mapped to 32-bit hash values, we need at least $\approx 2^{32(n+1)}$ hash functions: Multilinear is almost optimal.

Hence, the requirement to store many random numbers cannot be waived without sacrificing strong universality. Note that Stinson's bound is not affected by manipulations such as treating a length n string of W-bit words as a length $n/2$ string of $2W$-bit words.

If multiplications are expensive and we have long strings, we can attempt to improve speed by reducing the number of multiplications by half [18, 19]:

$$h(s) = m_1 + \sum_{i=1}^{n/2} (m_{2i} + s_{2i-1})(m_{2i+1} + s_{2i}). \qquad (1)$$

Indeed, this new form follows from the fact that we can rewrite the scalar product $m_{2i}s_{2i} + m_{2i+1}s_{2i-1}$ as a single multiplication $(m_{2i} + s_{2i-1})(m_{2i+1} + s_{2i})$ minus two terms, one that does not depend on the string $(m_{2i}m_{2i+1})$ and one that does not depend on the random keys ($s_{2i-1}s_{2i}$). While this new form assumes that the number of characters in the string is even, we can simply pad the odd-length strings with an extra character with value zero. With variable-length strings, the padding to even length must follow the addition of a character value of one.

Could we reduce the number of multiplications further? Not in general: the computation of a scalar product between two vectors of length n requires at least $\lceil n/2 \rceil$ multiplications [20, Corollary 4]. However,

we could try to avoid generic multiplications altogether and replace them by squares [16]:

$$h(s) = m_1 + \sum_{i=1}^{n} (m_{i+1} + s_i)^2.$$

Indeed, squares can be sometimes be computed faster. Unfortunately, this approach fails in binary finite fields ($GF(2^L)$) because

$$\begin{aligned}(m_{i+1} + s_i)^2 &= m_{i+1}^2 + m_{i+1}s_i + m_{i+1}s_i + s_i^2 \\ &= m_{i+1}^2 + s_i^2\end{aligned}$$

since every element is its own additive inverse. Thus, we get

$$h(s) = m_1 + \sum_{i=1}^{n} m_{i+1}^2 + \sum_{i=1}^{n} s_i^2$$

which is a poor hash function (e.g., $h(\mathtt{ab}) = h(\mathtt{ba})$).

There are fast algorithms to compute multiplications [21, 22, 23] in binary finite fields. Yet these operations remain much slower than a native operation (e.g., a regular 32-bit integer multiplication). However, some recent processors have support for finite fields. In such cases, the penalty could be small for using finite fields, as opposed to regular integer arithmetic (see § 4 and § 5.4). (Though they are outside our scope, there are also fast techniques for computing hash functions over a finite field having prime cardinality [24].)

3. MAKING MULTILINEAR STRONGLY UNIVERSAL IN THE RING $\mathbb{Z}/2^K\mathbb{Z}$

On processors without support for binary finite fields, we can trade memory for speed to essentially get the same properties as finite fields on <u>some</u> of the bits using fast integer arithmetic. For example, Dietzfelbinger [15] showed that the family of hash functions of the form

$$h_{A,B}(x) = (Ax + B \mod 2^K) \div 2^{L-1}$$

where the integers $A, B \in [0, 2^K)$ and $x \in [0, 2^L)$ is strongly universal for $K > L - 1$. (To reduce the number of parentheses used, we adopt the convention that $Ax + B \mod 2^K \equiv (Ax + B) \mod 2^K$. The symbol $\div$ denotes integer division: $x \div y = \lfloor x/y \rfloor$ for positive integers.) We generalize Dietzfelbinger hashing from the linear to the multilinear case.

The main difference between a finite field and common integer arithmetic (in the integer ring $\mathbb{Z}/2^K\mathbb{Z}$) is that elements of fields have inverses: given the equation $ax = b$, there is a unique solution $x = a^{-1}b$ when $a \neq 0$. However, the same is "almost" true in integer rings used for computer arithmetic as long as the variable a is small. For example, when $a = 1$, we can solve for $ax = b$ exactly ($x = b$). When $a = 2$, then there are at most two solutions to the equation

$ax = b$. We build on these observations to derive a stronger result.

We let $\tau = \text{trailing}(a)$ be the number of trailing zeros of the integer a in binary notation. For example, we have that $\text{trailing}(2^j) = j$.

PROPOSITION 3.1. *Given integers K, L satisfying $K \geq L - 1 \geq 0$, consider the equation*

$$(ax + c \mod 2^K) \div 2^{L-1} = b$$

where a is an integer in $[1, 2^L)$, b is an integer in $[0, 2^{K-L+1})$ and c an integer in $[0, 2^K)$. Given a, b and c, there are exactly 2^{L-1} integers x in $[0, 2^K)$ satisfying the equation.

Proof. Let $\tau = \text{trailing}(a)$. We have $\tau \leq L - 1$ since $a \in [1, 2^L)$. Because a is non-zero, we have that $a' = a \div 2^\tau$ is odd and $a = 2^\tau a'$.

We have

$$\begin{aligned}&((ax + c) \mod 2^K) \div 2^{L-1} \\ &\quad = ((2^\tau a'x + c) \mod 2^K) \div 2^{L-1} \\ &\quad = (2^\tau [a'x + (c \div 2^\tau)] \\ &\qquad + (c \mod 2^\tau) \mod 2^K) \div 2^{L-1}.\end{aligned}$$

We show that the term $(c \mod 2^\tau)$ can be removed. Indeed, consider that the τ least significant bits of $2^\tau[a'x + (c \div 2^\tau)] + (c \mod 2^\tau)$ are those of $c \mod 2^\tau$, and the more significant bits are those of $2^\tau[a'x + (c \div 2^\tau)]$. The final division by 2^{L-1} will dismiss the $L - 1$ least significant bits, and $\tau \leq L - 1$, so that the term $(c \mod 2^\tau)$ can be ignored.

Hence, we have

$$\begin{aligned}&(ax + c \mod 2^K) \div 2^{L-1} \\ &\quad = (2^\tau[a'x + (c \div 2^\tau)] \mod 2^K) \div 2^{L-1} \\ &\quad = (2^\tau [a'x + (c \div 2^\tau) \mod 2^{K-\tau}]) \div 2^{L-1} \\ &\quad = (a'x + (c \div 2^\tau) \mod 2^{K-\tau}) \div 2^{L-1-\tau} \\ &\quad = (a'(x \mod 2^{K-\tau}) \\ &\qquad + (c \div 2^\tau) \mod 2^{K-\tau}) \div 2^{L-1-\tau}.\end{aligned}$$

Setting $x' = x \mod 2^{K-\tau}$ and $c' = c \div 2^\tau$, we finally have

$$\begin{aligned}&(ax + c \mod 2^K) \div 2^{L-1} \\ &\quad = (a'x' + c' \mod 2^{K-\tau}) \div 2^{L-1-\tau}.\end{aligned}$$

Let z be an integer such that $z \div 2^{L-1-\tau} = b$. Consider $a'x' + c' \mod 2^{K-\tau} = z$. We can rewrite it as $a'x' \mod 2^{K-\tau} = z - c' \mod 2^{K-\tau}$. Because a' is odd, a' and $2^{K-\tau}$ are coprime (their greatest common divisor is 1). Hence, there is a unique integer $x' \in [0, 2^{K-\tau})$ such that $a'x' \mod 2^{K-\tau} = z - c' \mod 2^{K-\tau}$ [25, Cor. 31.25].

Given b, there are $2^{L-1-\tau}$ integers z such that $z \div 2^{L-1-\tau} = b$. Given x', there are 2^τ integers x in $[0, 2^K)$ such that $x' = x \mod 2^{K-\tau}$. It follows that there are

$2^{L-1-\tau} \times 2^{\tau} = 2^{L-1}$ integers x in $[0, 2^K)$ such that $\left(a'x' + c' \mod 2^{K-\tau}\right) \div 2^{L-1-\tau} = b$ holds. $\qquad \square$

EXAMPLE 1. Consider, for instance, the equation $(6x + 10 \mod 64) \div 4 = 5$. By Proposition 3.1, there must be exactly 4 solutions to this equation (setting $K = 6, L = 3$). We can find them using the proof of the lemma. The integer 6 has 1 trailing zero in binary notation (110) so that $\tau = 1$. We can write $6 = 2 \times 3$ so that $a' = 3$. Similarly, $c' = 10 \div 2 = 5$. Hence we must consider the equation $3x' + 5 \mod 2^5 = z$ for values of z such that $z \div 2 = 5$. There are two such values: $z = 10$ and $z = 11$. We have that

$$3x' + 5 \mod 32 = 10 \Rightarrow 3x' \mod 32 = 5 \Rightarrow x' = 23;$$
$$3x' + 5 \mod 32 = 11 \Rightarrow 3x' \mod 32 = 6 \Rightarrow x' = 2.$$

It remains to solve for x in $x' = x \mod 32$ with the constraint that x is an integer in $[0, 64)$. When $x' = 2$, we have that $x \in \{2, 34\}$. When $x' = 23$, we have that $x \in \{23, 55\}$. Hence, the solutions are 2, 23, 34 and 55.

Using Proposition 3.1, we can show that fast variations of Multilinear are strongly universal even though we use regular integer arithmetic, not finite fields.

THEOREM 3.1. *Given integers K, L satisfying $K \geq L - 1 \geq 0$, consider the families of $(K - L + 1)$-bit hash functions*

- MULTILINEAR:

$$h(s) = \left(\left(m_1 + \sum_{i=1}^{n} m_{i+1}s_i\right) \mod 2^K\right) \div 2^{L-1}$$

- MULTILINEAR-HM:

$$h(s) = \left(\left(m_1 + \sum_{i=1}^{n/2} (m_{2i} + s_{2i-1})(m_{2i+1} + s_{2i})\right) \right.$$
$$\left. \mod 2^K\right) \div 2^{L-1}$$

which assumes that n is even.

Here the m_i's are random integers in $[0, 2^K)$ and the string characters s_i are integers in $[0, 2^L)$. These two families are strongly universal over fixed-length strings, or over variable-length strings that do not end with the zero character. We can apply the second family to strings of odd length by appending an extra zero element so that all strings have an even length.

Proof. We begin with the first family (MULTILINEAR). Given any two distinct strings s and s', consider the equations $h(s) = y$ and $h(s') = y'$ for any two hash values y and y'. Without loss of generality, we can assume that the strings have the same length. If not, we can pad the shortest string with zeros without changing its hash value. We need to show that $P(h(s) = y \land h(s') = y') = 2^{2(L-K-1)}$. Because the two strings

are distinct, we can find j such that $s_j \neq s'_j$. Without loss of generality, assume that $s'_j - s_j \in [0, 2^L)$.

We want to solve the equations

$$\left(\left(m_1 + \sum_{i=1}^{n} m_{i+1}s_i\right) \mod 2^K\right) \div 2^{L-1} = y, \quad (2)$$
$$\left(\left(m_1 + \sum_{i=1}^{n} m_{i+1}s'_i\right) \mod 2^K\right) \div 2^{L-1} = y' \quad (3)$$

for integers $m_1, m_2, \ldots$ in $[0, 2^K)$.

Consider the following equation

$$\left(m_1 + \sum_{i=1}^{n} m_{i+1}s_i\right) \mod 2^K = z.$$

There is a bijection between m_1 and $z \in [0, 2^K)$. That is, for every value of m_1, there is a unique z, and vice versa. Specifically, we have

$$m_1 = z - \sum_{i=1}^{n} m_{i+1}s_i \mod 2^K.$$

If we choose z such that $z \div 2^{L-1} = y$, we effectively solve Equation 2. By substitution in Equation 3, we have

$$\left(m_{j+1}(s'_j - s_j) + z + \sum_{i \neq j, i=1}^{n} m_{i+1}(s'_i - s_i)\right.$$
$$\left. \mod 2^K\right) \div 2^{L-1} = y'.$$

This equation is independent of m_1. By Proposition 3.1, there are exactly 2^{L-1} solutions m_{j+1} to this last equation. (Indeed, in the statement of Proposition 3.1, substitute m_{j+1} for x, $s'_j - s_j$ for a, $z + \sum_{i \neq j, i=1}^{n} m_{i+1}(s'_i - s_i) \mod 2^K$ for c and y' for b.)

Meanwhile, there are 2^{L-1} possible values z such that $z \div 2^{L-1} = y$. Because there is a bijection between m_1 and z, there are also 2^{L-1} possible values for m_1.

So, focusing only on m_1 and m_{j+1}, there are $2^{L-1} \times 2^{L-1}$ values satisfying $h(s) = y$ and $h(s') = y'$. Yet there are $2^K \times 2^K$ possible pairs m_1, m_{j+1}. Thus the probability that $h(s) = y$ and also that $h(s') = y'$ is $\frac{2^{L-1} \times 2^{L-1}}{2^K \times 2^K} = 2^{2(L-K-1)}$, which completes the proof for the first family.

The proof that the second family (MULTILINEAR-HM) is strongly universal is similar. As before, set z in $[0, 2^K)$ such that $z \div 2^{L-1} = y$. Solve for m_1 from the first equation:

$$m_1 = \left(z - \sum_{i=1}^{n/2} (m_{2i} + s_{2i-1})(m_{2i+1} + s_{2i})\right) \mod 2^K.$$

Then by substitution, we get

$$\left(\left(\sum_{i=1}^{n/2} (m_{2i} + s'_{2i-1})(m_{2i+1} + s'_{2i}) - \right.\right.$$
$$(m_{2i} + s_{2i-1})(m_{2i+1} + s_{2i})$$
$$\left.\left. + z\right) \mod 2^K\right) \div 2^{L-1} = y'.$$

We can rewrite this last equation—if j is even, as $((m_j(s'_j - s_j) + \rho + z \mod 2^K) \div 2^{L-1} = y'$; if j is

odd, as $((m_{j+2}(s'_j - s_j) + \rho + z \mod 2^K) \div 2^{L-1} = y'$, where ρ is independent of either m_j (when j is even) or m_{j+2} (when j is odd). As before, by Proposition 3.1, there are exactly 2^{L-1} solutions for m_j (j even) or m_{j+2} (j odd) if z is fixed. As before, there are 2^{L-1} distinct possible values for z, and 2^{L-1} distinct corresponding values for m_1. Hence, the pair m_1, m_j can take $2^{L-1} \times 2^{L-1}$ distinct values out of $2^K \times 2^K$ values, which completes the proof. $\square$

To apply Theorem 3.1 to variable-length strings, we can append the character value one to all strings so that they never end with the character value zero, as in § 2. If we use Multilinear-HM, we should add the character value one before padding odd-length strings to an even length.

Theorem 3.1 is both more general (because it includes strings) and more specific (because the cardinality of the set of hash values is a power of two) than a similar result by Dietzfelbinger [15, Theorem 4]. However, we believe our proof is more straightforward: we mostly use elementary mathematics.

While Dietzfelbinger did not consider the multilinear case, others proposed variations suited to string hashing. Pǎtraşcu and Thorup [26] state without proof that Multilinear-HM over strings of length two is strongly universal for $K = 64, L = 32$. They extend this approach to strings, taking characters two by two:

$$h(s) = \left(\left(\bigoplus_{i=1}^{n/2}(m_{3i-2} + s_{2i-1})(m_{3i-1} + s_{2i}) + m_{3i}\right) \mod 2^K\right) \div 2^L$$

where $\bigoplus$ is the bitwise exclusive-or operation and n is even. Unfortunately, their approach uses more operations and requires 50% more random numbers than Multilinear-HM. They also refer to an earlier reference [27] where a similar scheme was erroneously described as universal, and presented as folklore:

$$h(s) = \left(\left(\bigoplus_{i=1}^{n/2}(m_{2i+1} + s_{2i+1})(m_{2i+2} + s_{2i+2})\right) \mod 2^K\right) \div 2^L$$

where n is even. To falsify the universality of this last family, we can verify numerically that the strings $0, 0$ and $2, 6$ collide with probability $\frac{576}{4096} > \frac{1}{2^3}$, for $K = 6, L = 3$. In any case, we see no benefit to this last approach for long strings because Multilinear-HM is likely just as fast, and it is strongly universal.

3.1. Implementing Multilinear

If 32-bit values are required, we can generate a large buffer of 64-bit unsigned random integers m_i. The computation of either

$$h(s) = \left(m_1 + \sum_{i=1}^{n} m_{i+1} s_i \mod 2^{64}\right) \div 2^{32}$$

or

$$h(s) = \left(m_1 + \sum_{i=1}^{n/2}(m_{2i} + s_{2i-1})(m_{2i+1} + s_{2i}) \mod 2^{64}\right) \div 2^{32}$$

is then a simple matter using unsigned integer arithmetic common to most modern processors. The division by 2^{32} can be implemented efficiently by a right shift (>>32).

One might object that according to Theorem 3.1, 63-bit random numbers are sufficient if we wish to hash 32-bit characters to a 32-bit hash value. The division by 2^{32} should then be replaced by a division by 2^{31}. However, we feel that such an optimization is unlikely to either save memory or improve speed.

Multilinear is essentially an inner product and thus can benefit from multiply-accumulate CPU instructions: by processing the multiplication and the subsequent addition as one machine operation, the processor may be able to do the computation faster than if the computations were done separately. Several processors have such integer multiply-accumulate instructions (ARM, MIPS, Nvidia and PowerPC). Comparatively, we do not know of any multiply-xor-accumulate instruction in popular processors.

Unfortunately, some languages—such as Java—fail to support unsigned integers. With a two's complement representation, the de facto standard in modern processors, additions and multiplications give identical results, up to overflow flags, as long as no promotion is involved: e.g., multiplying 32-bit integers using 32-bit arithmetic, or 64-bit integers using 64-bit arithmetic. However, we must still be careful: promotions and divisions differ when we use signed integers:

- If we store string characters using 32-bit integers (int) and random values as 64-bit integers (long), Java will sign-extend the 32-bit integer to a 64-bit integer when computing $m_{i+1} * s_i$, giving an unintended result for negative string characters. Use $m_{i+1} * (s_i \& 0xFFFFFFFFl)$ instead.
- Unsigned and signed divisions differ. Correspondingly, for the division by 2^{32}—to retrieve the 32 most significant bits—the unsigned right-shift operator (>>>) must be used in Java, and not the regular right shift (>>).

Because we assume that the number of bits is a constant, the computational complexity of Multilinear is linear $O(n)$. Multilinear uses n multiplications, n additions, and one shift, whereas Multilinear-HM uses $n/2$ multiplications, $3n/2$ additions, and one shift.

In both cases we use $2n + 1$ operations, although there may be benefits to having fewer multiplications. (Admittedly, Single Instruction, Multiple Data (SIMD) processors can do several instructions at once, making an analysis based solely on the number of operations misleading.)

Consider that we need at least $\approx 32(n + 1)$ random bits for strongly universal 32-bit hashing of $32n$ bits [17] according to Stinson's bound. That is, we must aggregate $\approx 64n + 32$ bits into a 32-bit hash value. Assume that we only allow unary and binary operations. A 32-bit binary operation maps 64 bits to 32 bits, a reduction of 32 bits. Hence, we require at least $2n$ 32-bit operations for strongly universal hashing. Alternatively, we require at least n 64-bit operations. Hence, for n large, Multilinear and Multilinear-HM use at most twice the minimal number of operations.

3.2. Word size optimization

The number of required bits is application dependent: for a hash table, one may be able to bound the maximum table size. In several languages such as Java, 32-bit hash values are expected. Meanwhile the key parameters of our hash functions Multilinear and Multilinear-HM are L (the size of characters) and K (the size of the operations), and these two hash functions deliver $K - L + 1$ usable bits.

However, both K and L can be adjusted given a desired number of usable random bits. Indeed, a length n string of L-bit characters can be reinterpreted as a length $n\lceil L/L' \rceil$ string of L'-bit characters, for any non-zero L'. Thus, we can either grow L and K or reduce L and K, for the same number of usable bits.

To reduce the need for random bits, we should use large values of K. Consider a long input string that we can represent as a string of 32-bit or 96-bit characters. Assume we want 32-bit hash values. Assume also that our random data only comes in strings of 64-bit or 128-bit characters. If we process the string as a 32-bit string, we require 64 random bits per character. The ratio of random strings to hashed strings is two. If we process the string as a 96-bit string, we require 128 random bits per character and the ratio of random strings to hashed strings is $128/96 = 4/3 \approx 1.33$. What if we could represent the string using 224-bit characters and have random bits packaged into characters of 256 bits? We would then have a ratio of $8/7 \approx 1.14$.

We can formalize this result. Suppose we require z pairwise independent bits and that we have M input bits. Stinson [17] showed that this requires at least $1 + 2^M(2^z - 1)$ hash functions. Equivalently, this requires $\log(1 + 2^M(2^z - 1))$ random bits. Thus, given any hashing family, the ratio of its required number of random bits to the Stinson limit (henceforth Stinson ratio) must be greater or equal to one. The M input bits can be represented as an L-bit n-character string for $M = nL$. Under Multilinear (and Multilinear-
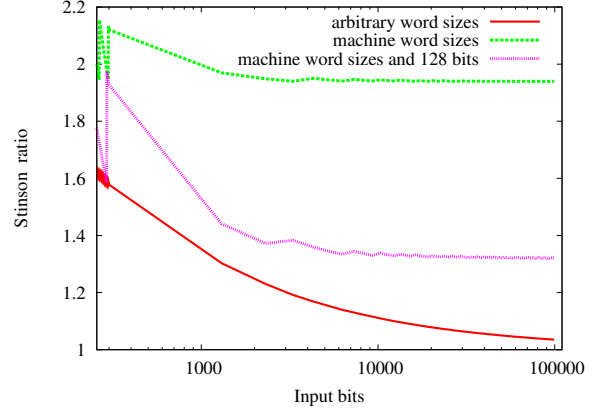

FIGURE 1: For large inputs, Multilinear requires an almost optimal number of random bits when arbitrary word sizes (K) are allowed. It has lower efficiency when the word size is constrained. The plot was generated for 32-bit hash values $(z = 32)$.

HM), we must have $z = K - L + 1$. Thus we use $K(n + 1) = (z + L - 1)(\lceil M/L \rceil + 1)$ random bits. We have that $(z + L - 1)(\lceil M/L \rceil + 1) \le (z + L - 1)(M/L + 2)$ which is minimized when

$$L = \sqrt{(z - 1)\frac{M}{2}}. \qquad (4)$$

Rounding $L = \sqrt{(z-1)M/2}$ up and substituting it back into $(z + L - 1)(\lceil M/L \rceil + 1)$, we get an upper bound on the number of random bits required by Multilinear. This bound is nearly optimal when $\lceil M/L \rceil \approx M/L$, that is, when M is large. Unfortunately, this estimate fails to consider that word sizes are usually prescribed. For example, we could be required to choose $K \in \{8, 16, 32, 64\}$. That is, we have to choose $L \in \{9 - z, 17 - z, 33 - z, 65 - z\}$. Fig. 1 shows the corresponding Stinson ratios. When there are many input bits $(M \gg 1)$, the ratio of Multilinear converges to one. That is, as long as we can decompose input data into strings of large characters (having approximately $\sqrt{(z-1)M/2}$ bits), Multilinear requires almost a minimal number of bits. This may translate into an optimal memory usage. (The result also holds for Multilinear-HM except that it is slightly less efficient for strings having an odd number of characters.) If we restrict the word sizes to common machine word sizes $(K \in \{8, 16, 32, 64\})$, the ratio is ≈ 2 for large input strings. We also consider the case where we could use 128-bit words (with $K \in \{8, 16, 32, 64, 128\}$). It improves the ratio noticeably (≈ 1.33), as expected.

We can also choose the word size (K) to optimize speed. On a 64-bit processor, setting $K = 64$ would make sense. We can compare this default with two alternatives:

1. We can try to support much larger words using fast multiplication algorithms such as Karatsuba's.

We could merely try to minimize the number of random bits. However, this ignores the growing computation cost of multiplications over many bits, e.g., Karatsuba algorithm is in $\Omega(K^{1.58})$. For simplicity, suppose that the cost of K-bit multiplication costs K^a time for $a > 1$. To hash M bits, we require $\lceil M/L \rceil$ multiplications with MULTILINEAR. When we have long strings (i.e., $M \gg L$), we can simplify $\lceil M/L \rceil \approx M/L$. If we desire z-bit hash values, then we need to use multiplication on $K = z + L - 1$ bits. Thus, the processing cost can be (roughly) approximated as $\frac{M(z+L-1)^a}{L}$. Starting from $L = 1$, this function initially decreases to a minimum at

$$L = \frac{z-1}{a-1} \quad (5)$$

before increasing again as L^{a-1}. (When $a = 1.5$ and $z = 32$, we have $\frac{z-1}{a-1} = 62$.) See Fig. 2. Hence, while we can minimize the total number of random bits by using many bits per character (L large), we may want to keep L relatively small to take into account the superlinear cost of multiplications.

2. We can support 128-bit words on a 64-bit processor, with some overhead. (Recent GNU GCC compilers have the __uint128 type, as a C-language extension.) A single 128-bit multiplication may require up to three 64-bit multiplications. However, it processes more data: with $z = 32$ hashed bits, each 128-bit multiplication hashes 97 input bits. Comparatively, setting $K = 64$, we require a single 64-bit multiplication, but we process only 32 bits of data. (Formally, we could process 33 bits of data, but for convenient implementation, we process data in powers of two.) Hence, it is unclear which approach is faster: three 64-bit multiplications and 128 bits of random data to process 97 input bits, or a single 64-bit multiplication and 64 bits of random data, to process 33 input bits. However, the 128-bit approach will use 33% fewer random bits. Going to 256-bit word sizes would only reduce the number of random bits by 14%: using larger and larger words leads to diminishing returns.

We assess these two alternatives experimentally in § 5.5.

4. FAST MULTILINEAR WITH CARRY-LESS MULTIPLICATIONS

To help support fast operations over binary finite fields ($GF(2^L)$), AMD and Intel introduced the Carry-less Multiplication (CLMUL) instruction set [28]. If we are given the binary representations of two numbers, $a = \sum_{i=1}^{L} a_i 2^{i-1}$ and $b = \sum_{i=1}^{L} b_i 2^{i-1}$, the carry-less multiplication is given by $c = \sum_{i=1}^{2L-1} c_i 2^{i-1}$, where $c_i = \bigoplus_{j=1+i}^{2L-1} a_j b_{j-i}$. Henceforth, we write
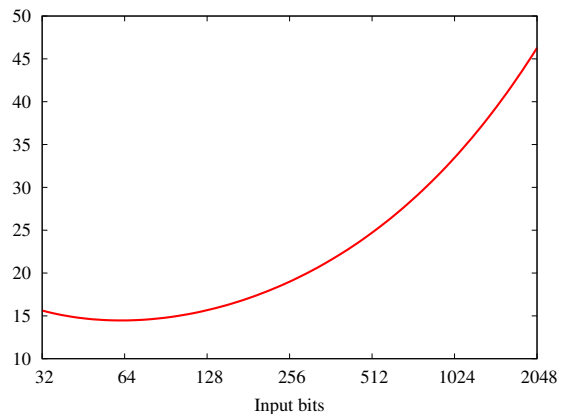

FIGURE 2: Modeled computational cost per bit as a function of the number of bits per character ($\frac{(z+L-1)^a}{L}$) for 32-bit hashing values ($z = 32$) and $a = 1.5$.

$a \star b = c$. If we represent the two L-bit integers a and b as polynomials in $GF(2)[x]$, then the carry-less multiplication is equivalent to the usual polynomial multiplication:

$$\left(\sum_{i=1}^{L} a_i x^{i-1} \right) \left(\sum_{i=1}^{L} b_i x^{i-1} \right) = \sum_{i=1}^{2L-1} c_i x^{i-1}.$$

With a fast carry-less computation, we can compute Multilinear efficiently. Given any irreducible polynomial $p(x)$ of degree L, the field $GF(2)[x]/p(x)$ is isomorphic to $GF(2^L)$. Hence, we want to compute $h(s) = m_1 + \sum_{i=1}^{n} m_{i+1} s_i$ over $GF(2)[x]/p(x)$. Computing all multiplications over $GF(2)[x]/p(x)$ would still be expensive given fast carry-less multiplication. Instead, we first compute $m_1 + \sum_{i=1}^{n} m_{i+1} s_i$ over $GF(2)[x]$ and then return the remainder of the division of the final result by $p(x)$. Indeed, think of the values $m_1, m_2, \ldots$ and $s_1, s_2, \ldots$ as polynomials of degree at most L in $GF(2)[x]$. Each of the n multiplications in $GF(2)[x]$ is equivalent to a carry-less multiplication over L-bit integers which results in a $2L - 1$-bit value. Similarly, each of the n additions in $GF(2)[x]$ is an exclusive-or operation. That is, we want to compute the $2L - 1$-bit integer

$$\bar{h}(s) = m_1 \oplus \left(\bigoplus_{i=1}^{n} m_{i+1} \star s_i \right). \quad (6)$$

Finally, considering $\bar{h}(s)$ as an element of $GF(2)[x]$, noted $q(x)$, we must compute $q(x)/p(x)$. The remainder (as an L-bit integer) is the final hash value $h(s)$.

If done naively, computing the remainder of the division by an irreducible polynomial may remain relatively expensive, especially for short strings since they require few multiplications. A common technique to quickly compute the remainder is the Barrett reduction algorithm [29]. The adaptation of this reduction to $GF(2)[x]$ is especially convenient [30] when

we choose the irreducible polynomial $p(x)$ such that degree$(p(x) - x^L) \leq L/2$, that is, when we can write it as $p(x) = \sum_{i=0}^{\lfloor L/2 \rfloor} a_i x_i + x^L$. (There are such irreducible polynomials for $L \in \{1, 2, \ldots, 400\}$ [31] and we conjecture that such a polynomial can be found for any L [32].) In this case, the remainder of $q(x)/p(x)$ is given by

$$((((q \div 2^L) \star p) \div 2^L) \star p) \oplus q) \mod 2^L$$

where q and p are the $2L - 1$-bit and $L + 1$-bit integers representing $q(x)$ and $p(x)$. (See Appendix B for implementation details.) We expect the two carry-less multiplications to account for most of the running time of the reduction. Yet we expect that even a fast implementation of the Barrett reduction is much slower than merely selecting the left-most L bits as in MULTILINEAR.

Unfortunately, in its current Intel implementation, carry-less multiplications have significantly reduced throughput compared to regular integer multiplications. Indeed, with pipelining, it is possible to complete one regular multiplication per cycle, but only one carry-less multiplication every 8 cycles [33]. However, using a result from § 2, we can reduce the number of multiplications by half if we compute

$$\bar{h}(s) = m_1 \oplus \left(\bigoplus_{i=1}^{n/2} (m_{2i} + s_{2i-1}) \star (m_{2i+1} + s_{2i}) \right)$$

instead. (Henceforth, we refer to this last variation as GF MULTILINEAR-HM, whereas we refer to the version based on Equation 6 as GF MULTILINEAR.)

However, irrespective of its speed, the carry-less approach might still be preferable to the schemes described in § 3 (e.g., MULTILINEAR) because fewer random bits are required. Indeed, to generate L-bit hash values from n-character strings, the carry-less scheme uses $(n + 1)L$ random bits, whereas MULTILINEAR requires $2L + n(2L - 1)$ random bits.

5. EXPERIMENTS

Our experiments show the following results:

- It is best to implement MULTILINEAR with loop unrolling. With this optimization, MULTILINEAR is just as fast (on Intel processors) as MULTILINEAR-HM, even though it has twice the number of multiplications. In general, processor microarchitectural differences are important in determining which method is fastest. (§ 5.2)
- In the absence of processor support for carry-less multiplication (see § 4), hashing using Multilinear over binary finite fields is an order of magnitude slower than MULTILINEAR even when using a highly optimized library. (§ 5.3)
- Even with hardware support for carry-less multiplication, hashing using Multilinear over binary finite

fields remains several times slower than MULTILINEAR. (§ 5.4)

- Given a 64-bit processor, it is noticeably faster to use a word size of 64 bits even though a larger word size (128 bits) uses fewer random bits (33% less). Use of multiprecision arithmetic libraries can further reduce the overhead from accessing random bits, but they also fail to be competitive with respect to speed, though they can halve the number of required random bits. (§ 5.5)
- MULTILINEAR is generally faster than popular string-hashing algorithms. (§ 5.6)

5.1. Experimental setup

We evaluated the hashing functions on the platforms shown in Table 1. Our software is freely available online [34]. For Intel and AMD processors, we used the processor's time stamp counter (`rdtsc` instruction [35]) to estimate the number of cycles required to hash each byte. Unfortunately, the ARM instruction set does not provide access to such a counter. Hence, for ARM processors (Apple A4 and Nvidia Tegra), we estimated the number of cycles required by dividing the wall-clock time by the documented processor clock rate (1 GHz).

For the 64-bit machines, 64-bit executables were produced and all operations were executed using 64-bit arithmetic except where noted. All timings were repeated three times. For the 32-bit processors, 32-bit operations were used to process 16-bit strings. Therefore, results between 32- and 64-bit processors are not directly comparable. Good optimization flags were found by a trial-and-error process. We note that using profile-guided optimizations did not improve this code any more than simply enabling loop unrolling (`-funroll-loops`). With (only) versions 4.4 and higher of GCC, it was sometimes important for speed to forbid use of SSE2 instructions when compiling MULTILINEAR and MULTILINEAR-HM (hence the `-mno-sse2` flags in Table 1). Moreover, we determined that versions 4.6 and 4.7 of GCC gave incorrect compiled code when vectorizing MULTILINEAR and related functions: as an alternative to the `-mno-sse2` flag, we found that the `-fno-tree-vectorize` flag was sufficient to ensure correct results.

We found that the speed is insensitive to the content of the string: in our tests we hashed randomly generated strings. We reuse the same string for all tests. Unless otherwise specified, we hash randomly generated 32-bit strings of 1024 characters.

In addition to MULTILINEAR and MULTILINEAR-HM we also implemented MULTILINEAR (2-by-2) which is merely MULTILINEAR with 2-by-2 loop unrolling. (See Appendix A for representative C implementations.)

Our timings should represent the best possible performance: the performance of a function may degrade [23] when it is included in an application

TABLE 1: Platforms used.

Processor	Bits	GCC version	Flags, besides -O3 -funroll-loops
64-bit processors			
Intel Core 2 Duo	64	GNU GCC 4.6.2	-march=core2 -mno-sse2
Intel Xeon X5260	64	GNU GCC 4.1.2	-march=nocona
Intel Core i7-860	64	GNU GCC 4.6.2	-march=corei7 -mno-sse2
Intel Core i7-2600	64	GNU GCC 4.6.3	-march=corei7-avx -ftree-no-vectorize
Intel Core i7-2677M	64	GNU GCC 4.6.2	-march=corei7 -mno-sse2
AMD Sempron 3500+	64	GNU GCC 4.4.3	-march=k8 -mno-sse2
AMD V120	64	GNU GCC 4.4.3	-march=amdfam10 -mno-sse2
AMD FX8150	64	GNU GCC 4.6.3	-march=bdver1 -ftree-no-vectorize
32-bit processors			
Intel Atom N270	32	GNU GCC 4.5.2	-march=atom
Apple A4	32	GNU GCC 4.2.1	-march=armv7
Nvidia Tegra 2	32	GNU GCC 4.4.3[a]	
VIA Nehemiah	32	GNU GCC 3.3.4	-march=i686

[a]From the Android NDK, configured for the android-9 platform, and used on a Motorola XOOM.

because of bandwidth and caching.

5.2. Reducing the multiplications or unrolling may fail to improve the speed

We ran our experiments over both the 32-bit and 64-bit processors. For the 32-bit processors, we generated both 16-bit and 32-bit hash values. Our experimental results are given in Table 2.

We see that over 64-bit Intel processors (except for the i7-2600), there is little difference between MULTILINEAR, MULTILINEAR (2-by-2) and MULTILINEAR-HM, even though MULTILINEAR and MULTILINEAR (2-by-2) have twice the number of multiplications. We believe that Intel processors use aggressive pipelining techniques well suited to these computations. On the AMD processors, MULTILINEAR-HM is the clear winner, being at least 33% faster.

For the 32-bit processors, we get vastly different results depending on whether we generate 16-bit or 32-bit hash values.

- As expected, it is roughly twice as expensive to generate 32-bit hash values than to generate 16-bit values.
- For the VIA processor, MULTILINEAR-HM is 45% faster than MULTILINEAR and MULTILINEAR (2-by-2). We suspect that the computational cost is tightly tied to the number of multiplications.
- When the 32-bit ARM-based processors generate 32-bit hash values, MULTILINEAR (2-by-2) is preferable. We are surprised that MULTILINEAR-HM is the worse choice. We believe that this is related to the presence of a multiply-accumulate instruction in ARM processors. When generating 16-bit hash values, MULTILINEAR (2-by-2) becomes the worse choice. There is no significant benefit to using MULTILINEAR-HM as opposed to MULTILINEAR.

- The Intel Atom processor benefits from MULTILINEAR-HM when generating 32-bit hash value, but MULTILINEAR is preferable to generate 16-bit hash values. As with the ARM-based processors, MULTILINEAR (2-by-2) is a poor choice for generating 16-bit hash values.

5.3. Binary-finite-field libraries are not competitive

We obtained the $\mathtt{mp}\mathbb{F}_b$ library from INRIA. This code is reported [36] to be generally faster than popular alternatives such as NTL and Zen, and our own tests found it to be more than twice as fast as Plank's library [37].

We computed Multilinear in $GF(2^{32})$, using the version with half the number of multiplications (see Equation 1) because the library does much more work in multiplication than addition. Even so, on a Core 2 Duo, hashing 32-bit strings of 1024 characters was an order of magnitude slower than MULTILINEAR: averaged over a million attempts, the code using $\mathtt{mp}\mathbb{F}_b$ required an average of $7.69\,\mu s$ per string, compared with $0.78\,\mu s$ for MULTILINEAR. While our implementation of MULTILINEAR uses twice as many random bits as Multilinear in $GF(2^{32})$, this gain is offset by the memory usage of the finite-field library.

5.4. Hardware-supported carry-less multiplications are not fast enough

Intel reports a throughput of one carry-less product every 8 cycles [33] on a processor such as the Intel Core i7-2600. Consider GF MULTILINEAR-HM: it uses one carry-less multiplication for every two 32-bit characters. Hence, it requires at least 4 cycles to process each character. Therefore, in the best scenario possible, GF MULTILINEAR-HM will be almost four times slower

TABLE 2: Estimated CPU cycles per byte for fast Multilinear hashing

	MULTILINEAR	2-by-2	MULTILINEAR-HM
64-bit processors and 32-bit hash values and characters			
Intel Core 2 Duo	0.54	**0.52**	**0.52**
Intel Xeon X5260	**0.50**	**0.50**	**0.50**
Intel Core i7-860	**0.42**	**0.42**	**0.42**
Intel Core i7-2600	0.35	**0.27**	0.28
Intel Core i7-2677M	0.25	**0.20**	**0.20**
AMD Sempron 3500+	0.63	0.60	**0.40**
AMD V120	0.63	0.63	**0.40**
AMD FX8150	0.88	1.00	**0.51**
64-bit arithmetic and 32-bit hash values and characters on 32-bit processors			
Intel Atom N270	4.2	4.2	**3.6**
Apple A4	3.0	**2.7**	3.3
Nvidia Tegra 2	3.3	**3.0**	4.9
VIA Nehemiah	12	12	**8.2**
32-bit processors and 16-bit hash values and characters			
Intel Atom N270	**2.1**	3.5	2.6
Apple A4	1.9	2.6	**1.7**
Nvidia Tegra 2	**1.8**	2.2	1.9
VIA Nehemiah	5.2	5.2	**3.6**

than MULTILINEAR-HM which requires only 1.1 cycles per 32-bit character (0.28 cycle per byte).

To assess the actual performance, we implemented both GF MULTILINEAR and GF MULTILINEAR-HM in C (§ Appendix B). We also implemented a variation on GF MULTILINEAR-HM (henceforth GF MULTILINEAR-HM-FAST) that loads data in blocks of four 32-bit integers.

- Of the Intel processors we tested, only the i7-2600 has support for the CLMUL instruction set. If we use the flags `-O3 -funroll-loops -corei7-avx`, we get 9.6 CPU cycles per 32-byte character with GF MULTILINEAR, 5.8 CPU cycles with GF MULTILINEAR-HM and only 4.3 CPU cycles with GF MULTILINEAR-HM-FAST. That is 2.4 cycles, 1.5 cycles and 1.1 cycles per byte respectively: about 4–9 times slower than MULTILINEAR-HM on the same platform (0.27 cycle per byte). We might be able to improve our implementation. However, the throughput of the carry-less multiplication limits the character throughput of GF MULTILINEAR and GF MULTILINEAR-HM to 8 and 4 cycles.
- One of our AMD processors (AMD FX8150) also supports the CLMUL instruction set. With the flags `-O3 -funroll-loops -march=bdver1`, it fares slightly better than the Intel counterpart: 6.8 CPU cycles per 32-byte character with GF MULTILINEAR, 4.1 CPU cycles with GF MULTILINEAR-HM and only 3.5 CPU cycles with GF MULTILINEAR-HM-FAST. That is 1.7 cycles, 1.0 cycle and 0.9 cycle per byte respectively. However, the same AMD processor can process each 32-bit character in 2.05 cycles
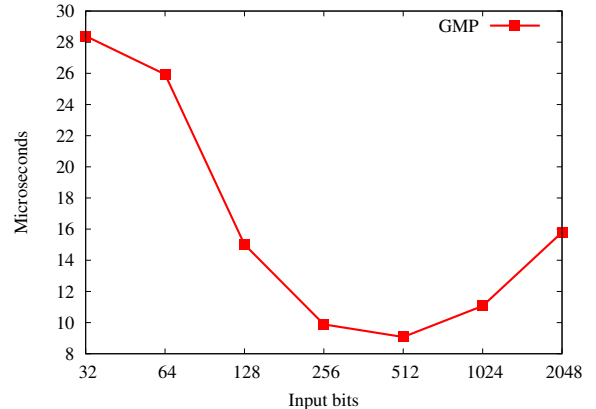

FIGURE 3: Microseconds to hash 4 kB using various word sizes and GMP.

(0.51 cycle per byte) with MULTILINEAR (2-by-2). Hence, MULTILINEAR (2-by-2) is almost 2 times faster than the best carry-less approach (GF MULTILINEAR-HM-FAST).

Overall, the hardware-supported carry-less Multilinear schemes are several times slower. On the bright side, GF MULTILINEAR and GF MULTILINEAR-HM require half the number of random bits.

5.5. The sweet-spot for multiprecision arithmetic is not sweet enough

To implement the techniques of § 3.2, we used the GMP library [38] version 5.0.2 to implement MULTILINEAR (2-by-2). As usual, we are hashing 4 kB of data, though data to be hashed are read in large chunks (up to 2048 bits). The hash output is always 32 bits ($z = 32$).

Results show a benefit as the chunk size L goes from 32 to 512 bits, but thereafter the situation degrades. See Fig. 3. In the best case, using 512-bit arithmetic, we require almost $13\,\mu s$ per string on a Core 2 Duo platform. For comparison, we find that the fewest random bits would be needed when $L = 1024$ (§ 3.2). As expected, the running time is minimized for a lower value of L to account for the superlinear cost of multiplication.

Unfortunately, we can do 12 times better without the GMP library ($0.78\,\mu s$ for 64-bit Multilinear), so it is not practical to use 512-bit arithmetic—even though it uses fewer random bits (nearly half as many).

As a lightweight alternative to a multiprecision library, we experimented with the `__uint128` type provided as a GCC extension for 64-bit machines. We used 128-bit random numbers and processed three 32-bit words with each 128-bit operation. Since `__uint128` multiplications are more expensive than `__uint128` additions, we tested the Multilinear-HM scheme. On the Core 2 Duo machine, the result was 38% slower than Multilinear (2-by-2) using 64-bit operations. This poor results is mitigated by the fact that we use 128 random bits per 96 input bits, versus 64 random bits per 32 input bits (a saving of nearly 33% for long strings). Investigation using hardware performance counters showed many "unaligned loads" from retrieving 128-bit quantities when we step through memory with 96-bit steps. To reduce this, we tried processing only two 32-bit words with each 128-bit operation, since we retrieved aligned 64-bit quantities. However, the result was 61% slower than Multilinear (2-by-2) using 64-bit operations.

5.6. Strongly universal hashing is inexpensive?

In a survey, Thorup [1] concluded that strongly universal hash families are just as efficient, or even more efficient, than popular hash functions with weaker theoretical guarantees. However, he only considered 32-bit integer inputs. We consider strings.

In Table 3, we compare the fastest Multilinear (Multilinear-HM) with two non-universal fast 32-bit string hash functions, Rabin-Karp [39] and SAX [40]. (They are similar to hash functions found in programming languages such as Java or Perl.) Even though these functions were designed for speed and lack strong theoretical guarantees, they are far slower than Multilinear on desktop processors (AMD and Intel). Only for ARM processors (Apple A4 and Nvidia Tegra 2) with 32-bit hash values are they much faster. We suspect that this good result on ARM processors is due to the multiply-accumulate instruction. Clearly, such a multiply-accumulate operation greatly benefits simple hashing functions such as Rabin-Karp and SAX.

Crosby and Wallach [11] showed that almost universal hashing could be as fast as common deterministic hash functions. One of their most competitive almost universal schemes is due to Black et al. [19]. Their fast family of hash functions is called NH:

$$h(s) = \sum_{i=1}^{n/2} \left(m_{2i-1} + s_{2i-1} \mod 2^{L/2} \right)$$
$$\times \left(m_{2i} + s_{2i} \mod 2^{L/2} \right) \mod 2^L.$$

NH is almost universal over fixed-length strings, or over variable-length strings that do not end with the zero character; we can apply it to strings having odd length by appending a character with value zero. It fails to be uniform: the value

$$(m_1 + s_1 \mod 2^{L/2})(m_2 + s_2 \mod 2^{L/2})$$

is zero whenever either $m_1 + s_1 \mod 2^{L/2}$ is zero or $m_2 + s_2 \mod 2^{L/2}$ is zero, which occurs with probability $\frac{2^{L/2+1}-1}{2^L} > \frac{1}{2^L}$ over all possible values of m_1, m_2. Moreover, the least significant bits may fail to be almost universal: e.g., for $L = 6$, there are 96 pairs of distinct strings colliding with probability 1 over the least two significant bits. When processing 32-bit characters, it generates 64-bit hash values with collision probability of $1/2^{32}$. Hence, in our tests over 32-bit characters, NH generates 64-bit hash values whereas the Multilinear families generate 32-bit hash values, but both have a collision probability bounded by $1/2^{32}$. Thus, while NH saves memory because it uses nearly half the number of random bits compared to our fast Multilinear families, Multilinear families may save memory in a system that stores hash values because their hash values have half the number of bits. Table 4 shows that the 64-bit NH on 64-bit processors runs at about the same speed as the best Multilinear on most processors. Only on two Intel Core i7 processors (2600 and 2677M), NH's running time is 60% of Multilinear's when we enable SSE support. On only one AMD processor (AMD FX8150), NH is 3 times faster. In other words, sacrificing theoretical guarantees does not always translate into better speed.

6. DISCUSSION

Overall, these numbers indicate that strongly universal string hashing is computationally inexpensive on most Intel and AMD processors. To get good results with older 64-bit and AMD processors, we recommend the use of Multilinear-HM. On more recent Intel processors (i7-2600 and i7-2677M), Multilinear (2-by-2) is just as fast.

Unfortunately—over long strings—strongly universal hashing requires many random numbers. Generating and storing these random numbers is the main difficulty. Whether this is a problem depends on the memory available, the CPU cache, the application workload and the length of the strings. (Intel researchers reported the generation of true random numbers in hardware at

TABLE 3: A comparison of estimated CPU cycles per byte between fast Multilinear hashing and common hash functions

	Rabin-Karp	SAX	best Multilinear
32-bit hash values and characters on 64-bit processors			
Intel Core 2 Duo	1.3	1.3	**0.52**
Intel Xeon X5260	1.4	1.6	**0.50**
Intel Core i7-860	1.4	1.6	**0.42**
Intel Core i7-2600	0.89	1.1	**0.27**
Intel Core i7-2677M	0.64	0.82	**0.20**
AMD Sempron 3500+	1.0	1.5	**0.40**
AMD V120	1.0	1.5	**0.40**
AMD FX8150	0.86	1.3	**0.51**
32-bit hash values and characters on 32-bit processors			
Intel Atom N270	**1.1**	2.0	4.2
Apple A4	**0.88**	1.2	2.7
Nvidia Tegra 2	**0.85**	1.2	3.0
VIA Nehemiah	**2.0**	3.0	8.2
16-bit hash values and characters on 32-bit processors			
Intel Atom N270	**2.1**	4.1	2.2
Apple A4	**1.8**	2.1	1.8
Nvidia Tegra 2	**1.6**	2.4	1.7
VIA Nehemiah	5.0	6.6	**3.6**

TABLE 4: A comparison of estimated CPU cycles per byte between fast Multilinear hashing and the almost universal hash function NH from Black et al. [19] for 32-bit hash values using 64-bit arithmetic. When running NH tests, we remove the `-mno-sse2` and `-fno-tree-vectorize` flags, where present, to get better results.

	NH [19]	best Multilinear
Intel Core 2 Duo	0.53	**0.52**
Intel Xeon X5260	**0.50**	**0.50**
Intel Core i7-860	**0.42**	**0.42**
Intel Core i7-2600	**0.16**	0.27
Intel Core i7-2677M	**0.12**	0.20
AMD Sempron 3500+	**0.38**	0.40
AMD V120	**0.38**	0.40
AMD FX8150	**0.17**	0.51

high speed (4 Gbps) [41].) In practice, unexpectedly long strings may require the generation of new random numbers while hashing a given string [11]. This overhead should be relatively inexpensive if we know the length of each string before we process it.

7. CONCLUSION

Over moderately long 32-bit strings ($\approx$1024 characters), current desktop processors can achieve strongly universal hashing with no more than 0.5 CPU cycle per byte, and sometimes as little as 0.2 CPU cycle per byte. Meanwhile, at least twice as many cycles are required for Rabin-Karp hashing even though it is not even universal.

While it uses half the number of multiplications, we have found that MULTILINEAR-HM is often no faster than MULTILINEAR on Intel processors. Clearly, Intel's pipelining architecture has some benefits. For AMD processors, MULTILINEAR-HM is faster ($\approx$ 33%), as expected because it uses fewer multiplications. Yet another alternative, MULTILINEAR (2-by-2), was slightly faster ($\approx$ 15%) for 32-bit hashing on the mobile ARM-based processors even though it requires twice as many multiplications as MULTILINEAR-HM. These mobile ARM-based processors also computed 32-bit Rabin-Karp hashing with fewer cycles per byte than many desktop processors. We believe that this is related to the presence of a multiply-accumulate in the ARM instruction set.

REFERENCES

[1] Thorup, M. (2000) Even strongly universal hashing is pretty fast. Proc. 11th ACM-SIAM Symp. on Discrete Algorithms, San Francisco, CA, 9–11 January, pp. 496–497. Society for Industrial and Applied Mathematics, Philadelphia.

[2] Ding, B. and König, A. C. (2011) Fast set intersection in memory. Proc. VLDB Endow., **4**, 255–266.

[3] Cormode, G. and Hadjieleftheriou, M. (2010) Methods for finding frequent items in data streams. VLDB J., **19**, 3–20.

[4] Gibbons, P. B. and Tirthapura, S. (2001) Estimating simple functions on the union of data streams. Proc. 13th ACM Symp. on Parallel Algorithms and Architectures, Heraklion, Crete Island, Greece, 4–6 July, pp. 281–291. ACM, New York.

[5] Bar-Yossef, Z., Jayram, T. S., Kumar, R., Sivakumar, D., and Trevisan, L. (2002) Counting distinct elements in a data stream. Proc. 6th Int. Workshop on Randomization and Approximation Techniques, Cambridge, MA, 13–15 September LNCS 2483, pp. 1–10. Springer-Verlag, Berlin.

[6] Pagh, R. and Rodler, F. (2004) Cuckoo hashing. J. Algorithms, **51**, 122–144.

[7] Dietzfelbinger, M. and Schellbach, U. (2009) On risks of using cuckoo hashing with simple universal hash classes. Proc. 13th ACM-SIAM Symp. on Discrete Algorithms, New York, NY, 4–6 January, pp. 795–804. Society for Industrial and Applied Mathematics, Philadelphia.

[8] Aumüller, M., Dietzfelbinger, M., and Woelfel, P. (2012) Explicit and efficient hash families suffice for cuckoo hashing with a stash. Proceedings of the 20th Annual European Conference on Algorithms, Ljubljana, Slovenia, 10–12 September LNCS 7501, pp. 108–120. Springer, Berlin.

[9] Carter, L. and Wegman, M. N. (1979) Universal classes of hash functions. J. Comput. Syst. Sci., **18**, 143–154.

[10] Wegman, M. and Carter, L. (1981) New hash functions and their use in authentication and set equality. J. Comput. Syst. Sci., **22**, 265–279.

[11] Crosby, S. A. and Wallach, D. S. (2003) Denial of service via algorithmic complexity attacks. Proc. 12th Conference on USENIX Security Symposium, Washington, DC, 4–8 August, pp. 3–3. USENIX, Berkeley.

[12] Klink, A. and WäldeAsanovic, J. (2011) Denial of service through hash table multi-collisions. Technical Report oCERT-2011-003. Open Source Computer Security Incident Response Team, Pittsburgh, PA.

[13] Zobrist, A. L. (1970) A new hashing method with application for game playing. Technical Report 88. Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. `http://www.cs.wisc.edu/techreports/viewreport.php?report=88`, last checked on 2013-05-13.

[14] Zobrist, A. (1990) A new hashing method with application for game playing. ICCA J., **13**, 69–73.

[15] Dietzfelbinger, M. (1996) Universal hashing and k-wise independent random variables via integer arithmetic without primes. Proc. 13th Symp. on Theoretical Aspects of Computer Science, Grenoble, France, 22–24 February LNCS 1046, pp. 569–580. Springer-Verlag, Berlin.

[16] Etzel, M., Patel, S., and Ramzan, Z. (1999) SQUARE HASH: Fast message authenication via optimized universal hash functions. Proc. 19th Int. Cryptology Conf. on Advances in Cryptology, Santa Barbara, CA, 15–19 August LNCS 1666, pp. 234–251. Springer-Verlag, Berlin.

[17] Stinson, D. R. (1994) Universal hashing and authentication codes. Des. Codes Cryptogr., **4**, 369–380.

[18] Motzkin, T. S. (1955) Evaluation of polynomials and evaluation of rational functions. Bull. Amer. Math. Soc., **61**, 163.

[19] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P. (1999) UMAC: Fast and secure message authentication. Proc. 19th Int. Cryptology Conf. on Advances in Cryptology, Santa Barbara, CA, 15–19 August LNCS 1666, pp. 216–233. Springer-Verlag, Berlin.

[20] Winograd, S. (1970) On the number of multiplications necessary to compute certain functions. Comm. Pure Appl. Math, **23**, 165–179.

[21] Brent, R. P., Gaudry, P., Thomé, E., and Zimmermann, P. (2008) Faster multiplication in $GF(2)[x]$. Proc. 8th Int. Conf. on Algorithmic Number Theory, Banff, Canada, 17–22 May LNCS 5011, pp. 153–166. Springer-Verlag, Berlin.

[22] Lemire, D. and Kaser, O. (2010) Recursive n-gram hashing is pairwise independent, at best. Comput. Speech Lang., **24**, 698–710.

[23] Greenan, K. M., Miller, E. L., and Schwarz, T. (2007) Analysis and construction of Galois fields for efficient storage reliability. Technical Report UCSC-SSRC-07-09. University of California, Santa Cruz, Santa Cruz, CA.

[24] Krovetz, T. and Rogaway, P. (2001) Fast universal hashing with small keys and no preprocessing: The PolyR construction. Proc. 3rd Int. Conf. Information Security and Cryptology, Seoul, Korea, 8-9 December 2000 LNCS 2015, pp. 73–89. Springer-Verlag, Berlin.

[25] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009) Introduction to Algorithms, Third Edition, 3rd edition. The MIT Press, Cambridge, MA.

[26] Pătraşcu, M. and Thorup, M. (2011) The power of simple tabulation hashing. Proc. 43rd ACM Symp. on Theory of Computing, San Jose, CA, 6-8 June, pp. 1–10. ACM, New York.

[27] Thorup, M. (2009) String hashing for linear probing. Proc. 20th ACM-SIAM Symp. on Discrete Algorithms, New York, NY, 4–6 January SODA '09, pp. 655–664. Society for Industrial and Applied Mathematics, Philadelphia.

[28] Gueron, S. and Kounavis, M. (2010) Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. Inform. Process. Lett., **110**, 549 – 553.

[29] Barrett, P. (1987) Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. Proc. Advances in Cryptology, Santa Barbara, CA, 18–22 August 1986 LNCS 263, pp. 311–323. Springer-Verlag, Berlin.

[30] Knežević, M., Sakiyama, K., Fan, J., and Verbauwhede, I. (2008) Modular reduction in $GF(2^n)$ without pre-computational phase. Proc. 2nd Int. Workshop on Arithmetic of Finite Fields, Siena, Italy, 6–9 July LNCS 5130, pp. 77–87. Springer, Berlin.

[31] Arndt, J. (2003). Minimal weight primitive polynomials over $GF(2)$. `http://www.jjj.de/mathdata/minweight-primpoly.txt`, last checked on 2013-05-13.

[32] Cohen, S. (2004) Primitive polynomials over small fields. _Proc. 7th Int. Conf. Finite Fields and Applications_, Toulouse, France, 5–9 May 2003 LNCS 2948, pp. 65–75. Springer, Berlin.

[33] 248966-026 (2011) _Intel 64 and IA-32 Architectures Optimization Reference Manual_. Intel Corporation. Santa Clara, CA.

[34] Kaser, O. and Lemire, D. (2012). A variable-length string hashing library in C. `http://code.google.com/p/variablelengthstringhashing/`, last checked on 2013-05-13.

[35] 324264-001 (2010) _How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures_. Intel Corporation. Santa Clara, CA.

[36] Gaudry, P. and Thomé, E. (2007) The $\mathrm{mp}\mathbb{F}_b$ library and implementing curve-based key exchanges. _Proc. of SPEED: Software Performance Enhancement for Encryption and Decryption_, Amsterdam, The Netherlands, 11–12 June, pp. 49–64. ECRYPT, Heverlee.

[37] Plank, J. S. (2007) Fast Galois field arithmetic library in C/C++. Technical Report UT-CS-07-593. University of Tennessee, Knoxville, TN. `http://web.eecs.utk.edu/~plank/plank/papers/CS-07-593.html`, last checked on 2013-05-13.

[38] Free Software Foundation (2011). The GNU MP Bignum library. `http://gmplib.org`, last checked on 2013-05-13.

[39] Karp, R. M. and Rabin, M. O. (1987) Efficient randomized pattern-matching algorithms. _IBM J. of Res. Dev._, **31**, 249–260.

[40] Ramakrishna, M. V. and Zobel, J. (1997) Performance in practice of string hashing functions. _Proc. Int. Conf. on Database Systems for Advanced Applications_, Melbourne, Australia, 1–4 April, pp. 215–224. World Scientific Press, Singapore.

[41] Srinivasan, S., Mathew, S., Erraguntla, V., and Krishnamurthy, R. (2009) A 4 Gbps 0.57 pJ/bit process-voltage-temperature variation tolerant all-digital true random number generator in 45 nm CMOS. _Proc. 22nd Int. Conf. on VLSI Design_, New Delhi, India, 5–9 January, pp. 301–306. IEEE Computer Society, Washington.

APPENDIX A. IMPLEMENTATIONS IN C

We implemented the following hash functions:

- MULTILINEAR:
  $h(s) = m_1 + \sum_{i=1}^{n} m_{i+1} s_i$

- MULTILINEAR (2-by-2):
  $h(s) = m_1 + \sum_{i=1}^{n/2} m_{2i} s_{2i-1} + s_{2i} m_{2i+1}$

- MULTILINEAR-HM:
  $h(s) = m_1 + \sum_{i=1}^{n/2} (m_{2i} + s_{2i-1})(s_{2i} + m_{2i+1})$

For simplicity we assume that the number of characters (n) is even. Following a common convention, we write the unsigned 32-bit and 64-bit integer data types as `uint32` and `uint64`. The variable `p` is a pointer to the initial value of the string whereas `endp` is a pointer to the location right after the last 32-bit character of the string. The variable `m` is a pointer to the 64-bit random numbers. (When using 63-bit random numbers as allowed by Theorem 3.1, the right shifts should be by 31 instead. In practice, we use 64-bit numbers.) On some compilers and processors, it was useful to disable SSE2: under GNU GCC we can achieve this result with function attributes (e.g. by preceding the function declaration by `__attribute__ ((__target__ ("no-sse2")))`).

MULTILINEAR

```
uint32 hash(uint64 * m, uint32 * p,
    uint32 * endp) {
  uint64 sum = *(m++);
  for (;p != endp; ++m, ++p)
    sum+= *m * *p;
  return sum >> 32;
}
```

MULTILINEAR (2-by-2)

```
uint32 hash(uint64 * m, uint32 * p,
    uint32 * endp) {
  uint64 sum = *(m++);
  for (; p != endp; m += 2, p += 2)
    sum += (*m * *p) + (*(m+1) * *(p+1));
  return sum >> 32;
}
```

MULTILINEAR-HM

```
uint32 hash(uint64 * m, uint32 * p,
    uint32 * endp) {
  uint64 sum = *(m++);
  for (; p != endp; m += 2, p += 2) {
    sum += (*m + *p) * (*(m+1) + *(p+1));
  }
  return sum >> 32;
}
```

APPENDIX B. CODE WITH CLMUL

We implemented Multilinear in $GF(2^{32})$ in C using the Carry-less Multiplication (CLMUL) instruction set [28] supported by recent Intel and AMD processors. We also implemented the counterpart to MULTILINEAR-HM which executes half the number of multiplications.

We use the same conventions as in Appendix A regarding the variables `p` and `m` except that the latter is a pointer to 32-bit random numbers. We wrote our C programs using SSE intrinsics: they are functions supported by several major compilers (including GNU GCC, Intel and Microsoft) that generate SIMD instructions.

The Barrett reduction algorithm is adapted from Knežević et al. [30]. The variable `C` contains the chosen

irreducible polynomial. We initialize it as

```
C = _mm_set_epi64x(0,1UL+ (1UL<<2)+ (1UL<<6)
    + (1UL<<7)+ (1UL<<32));.
```

Barrett reduction

```
uint32 barrett( __m128i A) {
  __m128i Q1 = _mm_srli_epi64 (A, n);
  __m128i Q2 = _mm_clmulepi64_si128( Q1, C,
      0x00);
  __m128i Q3 = _mm_srli_epi64 (Q2, n);
  __m128i f = _mm_xor_si128 (A,
    _mm_clmulepi64_si128( Q3, C, 0x00));
  return _mm_cvtsi128_si64(f) ;
}
```

GF Multilinear

```
uint32 hash(uint32 * m, uint32 * p,
  uint32 * endp) {
  __m128i sum = _mm_set_epi64x(0, *(m++));
  for (; p != endp; ++m, ++p ) {
    __m128i t = _mm_set_epi64x(*m, *p);
    __m128i c
    = _mm_clmulepi64_si128( t, t, 0x10);
    sum = _mm_xor_si128 (c, sum);
  }
  return barret(sum);
}
```

GF Multilinear-HM

```
uint32 hash(uint32 * m, uint32 * p,
    uint32 * endp) {
  __m128i sum = _mm_set_epi64x(0, *(m++));
  for (; p != endp; m += 2, p += 2 ) {
    __m128i t1 = _mm_set_epi64x(*m,*(m+1));
    __m128i t2 = _mm_set_epi64x(*p,*(p+1));
    __m128i t = _mm_xor_si128(t1, t2);
    __m128i c = _mm_clmulepi64_si128(t, t,
      0x10);
    sum = _mm_xor_si128 (c, sum);
  }
  return barret(sum);
}
```

GF Multilinear-HM-Fast

```
uint32 hash(uint32 * m, uint32 * p,
    uint32 * endp) {
 // assume m, p, endp are 128-bit aligned
  __m128i z = _mm_setzero_si128();
  __m128i sum = _mm_set_epi64x(0, *m);
  m += 4;
  __m128i t, u, t1, t2, ts, c1, c2;
  for (; p != endp; m += 4, p += 4 ) {
    t1 = _mm_load_si128((__m128i *) m);
    t2 = _mm_load_si128((__m128i *) p);
    ts = _mm_xor_si128(t1,t2);
    t = _mm_unpacklo_epi32(ts,z);
    c1 = _mm_clmulepi64_si128(t, t,0x10);
    sum = _mm_xor_si128(c1,sum);
    u = _mm_unpackhi_epi32(ts,z);
    c2 = _mm_clmulepi64_si128(u, u,0x10);
    sum = _mm_xor_si128 (c2,sum);
  }
  return barret(sum);
}
```