TNC Transactions on Networks and Communications
SOCIETY FOR SCIENCE AND EDUCATION UNITED KINGDOM
VOLUME 3, ISSUE 1
ISSN: 2054 - 7420

Software methods for fast hashing

Shay Gueron
University of Haifa, Israel;
Intel Corporation, Israel Development Center, Israel
shay@math.haifa.ac.il

ABSTRACT

The carry-less multiplication instruction, PCLMULQDQ, is a relatively recent addition to the x86-64 instructions set. It multiplies two binary polynomials of degree63, using $GF(2)$ arithmetic, and produces a polynomial of degree126, stored in a 128-bit register. PCLMULQDQ is intended to speed up computations in$GF(2^{128})$, which are used for AES-GCM authenticated encryption. We show here how PCLMULQDQ can be used for efficient software implementation of a 64-bit hash function that has a low collision probability. While a 64-bit hash is normally not a meaningful security primitive, the discussed hashing algorithm can be leveraged for other usages that enjoy fast hashing, e.g., querying/maintaining databases. On the latest Intel architecture (Codename Broadwell), our hash function can process messages at the rate of ~ 0.13 cycles per byte.

Keywords: hashing, universal hash functions, fast software implementations, PCLMUQDQ.

1 Introduction

Intel architectures introduced instructions that accelerate AES (AES-NI), and supplemented them with the carry-less multiplication instruction$PCLMULQDQ$. It was motivated by the desire to accelerate AES-GCM authenticated encryption, but other usages (e.g., CRC computations) were also considered (see details on $PCLMULQDQ$ and its applications in [5, 6]).

In this paper, we are interested in a non-cryptographic hashing algorithm that has a low collision probability, and can be computed efficiently. In particular, we seek a 64-bit hash. Hash functions with a relatively short digest have various usages, and a leading one is searching/updating databases. For such usages, the hashed messages are relatively short. Typical entries from database columns are, for example, zip code, name, address, salary, age, employer, and inventory. The associated message lengths are: 5 bytes (zip code), 10 bytes (telephone number), 32 bytes (city name).

There are many known (and used) hash functions. Two examples are LOOKUP3 [3], and Google's CityHash [2] (additional examples can be looked up, for example, in [1]). To the best of our knowledge, the collision probability for these hash functions is either verified empirically, and/or depends on some assumptions on the properties of the messages.

In this paper, we discuss a 64-bit hash function that has a provable low collision probability, and can be computed efficiently on modern computer platforms.

DOI: 10.14738/tnc.31.918
Publication Date: 26th February 2015
URL: http://dx.doi.org/10.14738/tnc.31.918

2 Preliminaries and notation

Let n be a positive integer, denotes $= n - 1$, and let $G = GF(2^n)/P(x)$ be the finite field with 2^n elements, represented via the irreducible polynomial $P(x)$ (of degree$n$). A field element $A \in G$ is a binary polynomial of degrees, $A = A(x) = \sum_{j=0}^{s} a_j\, x^j$ where $a_j \in \{0,1\}$. We also view A as the string of n bits, $a_s a_{s-1} \dots a_0$ (by convention a_s is at the leftmost position). For$A, B \in G$, $A = A(x) = \sum_{j=0}^{s} a_j\, x^j$ and$B = B(x) = \sum_{j=0}^{s} b_j\, x^j$, define addition and multiplication as follows:

1. Addition (+): $C = A + B$ is the polynomial$C = C(x) = \sum_{j=0}^{s} \left(\left(a_j + b_j\right)\ mod\ 2 \right) x^j$. The string representation of C is the bitwise XOR$A \oplus B$.
2. Multiplication ($\otimes$): $C(x) = A(x) \otimes B(x) = A(x) \times B(x)\ mod\ P(x)$, where "×" denotes polynomial multiplication with coefficients added/multiplied in $GF(2)$. Specifically, if $T(x) = A(x) \times B(x)$, then $T(x)$ is the binary polynomial of degree $2s$, with coefficients t_i satisfying $t_i = \left(\sum_{j=0}^{i} a_j b_{i-j}\right) mod\ 2$ for $0 \le i \le s$ and $t_i = \left(\sum_{j=i-s}^{s} a_j b_{i-j}\right) mod\ 2$ for $s < i \le 2s$.

When $n = 8m$, an element $F = F(x) = \sum_{j=0}^{s} f_j\, x^j \in G$ can be represented in a compact form, by a sequence of m bytes $B_{m-1} B_{m-2} \dots B_0$, and written in hexadecimal notation.

Hereafter, we focus on $n = 64$ and choose the irreducible polynomial $P(x) = x^{64} + x^4 + x^3 + x + 1$. We call a 64-bit string a "quadword".

To illustrate, we offer the following example.

Example 1: Use $n = 64$, $P(x) = x^{64} + x^4 + x^3 + x + 1$.
The element $F = x^{63} + x^4 + x^3 + x + 1$ is written in hexadecimal (compact) notation as the 8 bytes string $800000000000001B$. If $A = FFFFFFFF0000000F$, $B = FFFFFFFF0000010E$, then $A \times B = 55555555555555AA000000FF00000F5A$, and $A \otimes B = 000000FF00000615$.

2.1 The instruction $PCLMULQDQ$

The instruction $PCLMULQDQ$ operates on two 128-bit registers e.g., $xmm1$, $xmm2$ (the registers' names are arbitrary, and the second operand can also be in a specified in a memory location). It executes polynomial multiplication of two polynomials of degree 63, and writes the result (which is polynomial of degree 126) into a third 128-bit register (see precise definitions in [5]). An "immediate" byte selects the two 64-bit halves from $xmm1$ and $xmm2$, which are to be multiplied. For example, $VPCLMULQDQ\ \ xmm0,\ xmm1,\ xmm2,\ \ imm = 0x10$ operates on $xmm1[127:64]$ (top half of $xmm1$) and on $xmm2[63:0]$ (bottom half of $xmm2$), writing their carry-less product into $xmm0$. With A, B from Example 1 (embedded in two xmm registers), invoking $VPCLMULQDQ\ \ xmm0,\ xmm1,\ xmm2,\ \ 0x01$ with the registers contents $xmm1 = 0000000000000000FFFFFFFF0000000F$and
$xmm2 = FFFFFFFF0000010E0000000000000000$, results in the value $55555555555555AA000000FF00000F5A$ in $xmm0$.

3 Efficient hash functions with $PCLMULQDQ$

To define the message space, over which we use $PCLMULQDQ$ and carry out computations in $GF(2^{64})$, we need each message to be a string of quadwords. To this end, we pad messages of arbitrary lengths (in bytes) to the 8 bytes boundary, with zero bytes. In addition, to distinguish messages of variable lengths, we also append a quadword that encodes the message's length. The exact procedure is defined in the following subsections.

3.1 Message padding and length encoding

Consider a message MSG whose length, in bytes, is $lbytes$, and write $lbytes = 8u + v$, where $0 \leq v < 8$. We encode $lbytes$ as a 64-bit quadword, denoted LEN. For example, if $lbytes = 4096$, then $LEN = 0000000000001000$ (in hexadecimal notation).

If $v > 0$, we pad MSG up to the 8 bytes boundary, with $8 - v$ zero bytes. After the (conditional) padding, the padded message consists of $\ell_1 = \lceil lbytes/8 \rceil = u + 1$ quadwords, say $\overline{MSG} = X_1, X_2, \ldots, X_{\ell_1}$ (concatenated quadwords, in this order). Finally, we set

$$M = \overline{MSG}, LEN = X_1, X_2, \ldots, X_{\ell_1}, LEN$$

M consists of $\ell = \ell_1 + 1$ quadwords. We call M the "formatted message".

Example 2: Take a 9 bytes message $MSG = 090807060504030201$. Here, $lbytes = 9 = 8 \cdot 1 + 1$. This implies that $\ell_1 = \lceil 9/8 \rceil = 2$, and $X_1 = 0908070605040302$, $X_2 = 0100000000000000$. Also, $LEN = 0000000000000009$. Therefore,
$\overline{MSG} = 09080706050403020100000000000000$, and the formatter message is $M = \overline{MSG}, LEN = 0908070605040302, 0100000000000000, 0000000000000009$ (ℓ =3 quadwords).

3.2 The two hash functions

Let ℓ_{max} be a fixed value, and assume that the message space is the set of formatted messages of length $\ell \leq \ell_{max}$ quadwords. Let MSG be a message and let M be its formatted message, consisting of ℓ quadwords. Denote $M = X_1, X_2, \ldots, X_\ell$. Consider a key K that consists of ℓ_{max} quadwords $K = K_1, K_2, \ldots, K_{\ell_{max}}$ (each K_j is a quadword). We define two hash functions (S, T) for M, using the key K, as follows.

$$S = S(M, K) = \sum_{j=1}^{\ell} X_j \times K_j \qquad (S \in \{0,1\}^{128}) \qquad (1)$$

$$T = T(M, K) = \sum_{j=1}^{\ell} X_j \otimes K_j \qquad (T \in \{0,1\}^{64}) \qquad (2)$$

The function T is the well-known "inner product" XOR-universal hash function, defined over $(GF(2^{64}))^{\ell_{max}}$. Note that $T = S \ mod \ P(x)$.

3.3 Properties of S and T

Proposition 1. Suppose that the $K_{\ell_{max}}$ quadword keys $K_1, K_2, \ldots, K_{\ell_{max}}$ are selected independently, uniformly at random, from $\{0,1\}^{64}$. Let M be a (formatted) message of $\ell \leq \ell_{max}$ quadwords (not all of them are zero). Then,

$$Prob\ (S(M, K) = 0^{128}) < Prob\ (T(M, K) = 0^{64}) = \frac{1}{2^{64}} \qquad (3)$$

Remark 1. T is an "inner product" universal hash function (technically, the family of functions $T\ (M, K)$ is universal). Proposition 1 is a well-known property. The use of the quadword LEN, to account for the message length, is essential in order to ensure "suffix-freeness". To illustrate, note that without appending LEN, a message of a single quadword Q would have the same digest as the message of 2 quadwords $0000000000000000, Q$. This does not happen for the formatted messages. Since $T = S \ mod \ P(x)$, it follows that $S = 0 \Rightarrow \ T = 0$, so the number of messages that zero S is smaller than the number of messages that zero T.

Remark 2. Due to the linearity of S (and T), and to Proposition 1, the probability that two different messages M_1, M_2 have the save value $T = TS(M, K)$ is $\frac{1}{2^{64}}$.

Remark 3. Suppose that K_0 is chosen uniformly at random from $\{0,1\}^{64}$, and define $K_j = (K_0)^j \ mod \ P(x), j = 1, \ldots, \ell$. Then, for a nonzero message M of length ℓ quadwords, we have

$$Prob\ (S(M,K) = 0^{64}) \leq \frac{\ell}{2^{64}}$$

This bound is obtained by the following argument. S is an evaluation of a single variable polynomial in the field , whose degree is ℓ (K_0 is it variable). As such, it can have at most ℓ roots in the field.

3.4 The computational cost of computing S and T

Software running on processors that support $PCLMULQDQ$, can compute S by means of ℓ invocations of $PCLMULQDQ$, followed by no more than $\ell - 1$ XOR operations (depending on the method for XOR-ing the intermediate products; note that the $PXOR$ instruction can XOR 128-bit strings in a single 1-cycle invocation). To compute T, it suffices to compute S, and subsequently reduce it modulo $P(x)$.

3.4.1 Estimating the computational cost of computing S

Suppose that the instruction $PCLMULQDQ$ has latency of L cycles, and throughput 1 (the generalization to a different throughput is straightforward). Throughput 1 implies that the processor is capable of dispatching a $PCLMULQDQ$ instruction every cycle (if data to feed the instruction is available). Code that computes S can be written so that its flow interleaves (as much as it can) the operations and assures that data is available to feed $PCLMULQDQ$ every cycle. Consequently, it is theoretically possible to compute ℓ polynomial multiplications ("× ") in $\ell + L - 1$ cycles. Since the processors that we discuss here can execute two $PXOR$ operations in parallel to $PCLMULQDQ$, we neglect the additional overhead of the XOR operations required for the summation. We conclude that the rate at which the ℓ back-to-back "×" multiplications can be executed is $\frac{\ell+L-1}{8\ell}$ cycles per byte (C/B hereafter), which approaches $\frac{1}{8}$ C/B for large ℓ. Therefore, the best throughput for computing S, which we can expect, is 0.125 C/B. To illustrate, note that the latency of $PCLMULQDQ$ is $L = 7$, and the throughput is 1, on the latest Intel architecture Codename Broadwell (BDW hereafter), so $\frac{\ell+L-1}{8\ell} \sim 0.148$ C/B already for $\ell = 32$. The results shown in Section 4, indicate that the theoretical performance can be approached in practice.

3.4.2 Estimating the computational cost of computing S

Deriving T from S requires one reduction step (modulo $P(x)$). An efficient algorithm can carry out the reduction by means of 2 $PCLMULQDQ$ invocations, and it is demonstrated (as a real code snippet) in Listing 1 of the Appendix. On BDW, this reduction algorithm consumes (only) ~ 17 cycles.

A code example for computing S (and T) is provided in Listing 2 of the Appendix.

3.4.3 Using the hash functions

For a real application, it is reasonable to assume that an upper bound on the message length (thus on ℓ_{max}) is known in advance. Therefore, an application that needs to compute (efficiently) many hashes, can generate, as a setup phase, ℓ_{max} keys, and hold them in memory for the computations. This involves generating, uniformly at random, $64 \cdot \ell_{max}$ bits, but the cost is amortized over many hash computations. Often, the computed hash digests are ephemeral, so the keys do not need to be stored (to a disk) and retrieved in subsequent sessions. However, in cases they need to be stored, it is possible to fix one (randomly selected) seed ($seed$) and derive the keys by using some pseudo-random function. One example is to define $K_j = AES_{seed}(j)$. An alternative is to use powers (in the

field) of a single key, as in Remark 3, but this comes at the expense a larger bound (depending on ℓ_{max}) for the collision probability.

4 Results

To test the efficiency of our hash functions, we wrote optimized code and measured its performance on the following three latest Intel processors: Architecture Codenames Sandy Bridge (SNB), Haswell (HSW) and Broadwell (BDW).

In particular, we investigated the effect on the observed performance of the hash function, of the different latency and throughput of $PCLMULQDQ$. The relevant $PCLMULQDQ$ latency/throughput values on these processors are: $14/8$ cycles in SNB, $7/2$ cycles in HSW, and $7/1$ cycles in BDW.

The performance results are reported in Table 1. It shows the performance of computing S and T for short (from 1 bytes) messages up to 4KB messages, where the performance is already at its asymptotic value. For large messages, we see that the performance approaches its theoretical limit. For example, the reported code achieves 0.13 C/B on BDW, where the theoretical limit is 0.125 C/B. The small gap between the achieved and the theoretical performance, can be attributed to overheads such as, for example, function calls, data movement/alignment, and pointers arithmetic. As expected, the performance on HSW and on SNB is almost linearly proportional to the PCLMULQDQ throughput.

For long messages, we see that the difference between computing S and T is negligible. However, for short message, the reduction overhead is noticeable, and leads the different costs for computing S and versus computing T.

To test the collision properties, we wired the function to the Google Murmurhash test harness in [4] (that tries to “challenge” hash functions). As expected, we did not find any collisions in a millions of tests.

Table 1. The performance of computing S and T on three different architectures.

$lbytes$		1	5	10	16	20	32	64	128	1024	4096
	Cycles										
SNB	S	70	73	77	59	87	78	115	184	1,179	4,588
	T	46	49	53	39	61	59	97	167	1,161	4,571
HSW	S	50	54	54	18	57	22	29	45	270	1,053
	T	34	38	38	16	41	18	24	38	263	1,044
BDW	S	48	52	49	16	54	20	21	28	140	541
	T	34	38	36	14	40	17	19	27	139	541
	C/B										
SNB	S	69.68	14.58	7.72	3.68	4.34	2.43	1.79	1.44	1.15	1.12
	T	45.52	9.86	5.29	2.42	3.07	1.83	1.51	1.31	1.13	1.12
HSW	S	50.32	10.84	5.42	1.15	2.85	0.70	0.46	0.35	0.26	0.26
	T	34.32	7.66	3.84	1.01	2.07	0.58	0.38	0.30	0.26	0.26
BDW	S	48.32	10.46	4.93	1.01	2.70	0.63	0.33	0.22	0.14	0.13
	T	34.36	7.64	3.58	0.89	2.02	0.54	0.30	0.21	0.14	0.13

5 Discussion

This paper discussed two very simple hash functions that have two important properties: 1) They have a low collision probability; 2) On the modern processors, they have efficient software implementations. These hash functions can be useful in applications that manage databases.

For comparison, we point out that it is possible to compute 32-bit and 64-bit CRC's with $PCLMULQDQ$, and to approach the theoretical performance limit of 0.125 C/B (for large messages). However, the collision rate characteristics of such hashing strategies are completely different from what we have for S and T (and depend on what is assumed on the messages). Indeed, testing verified that a CRC hashing strategy stumbles on collisions for the same sets of messages that were used for challenging S and T (where we found no collisions).

The use of $S \in \{0,1\}^{128}$ (instead of $T \in \{0,1\}^{64}$) is more efficient for very short messages, and this can be considered as the preferable approach in such cases. This approach trades the improved performance with extra storage for longer hash digests. We point out that simply truncating S to 64 bits also gives a 64-bit hash. However, the bound on the collision probability is no longer guaranteed with this approach.

We conclude with two options to achieve further optimization for short message.

1. A single reduction step (see Listing 1 of the Appendix) consumes (on BDW) ~ 17 cycles because the latency of $PCLMULQDQ$ is 7 cycles. However, the effect of $PCLMULQDQ$'s latency can be significantly reduced for cases that require computing hashes of multiple messages, hence

involving multiple reductions. This can be achieved by aggregating multiple hash computations (and reductions), and interleaving the operations in the software flow, to get efficient pipelined execution.

2. Note that our padding method pads a message to the 8 bytes boundary, and then appends an additional quadword that encodes the message length.

    1. The use of the LEN quadword can be avoided completely if an application requires hashing messages with a fixed length.

    2. For variable length short messages, we can "compress" the length encoding. For example, suppose that all the messages are shorter than 256 bytes (which is a reasonable assumption for databases). Then, the lengths of hashed messages can be encoded by a single byte. Now, consider a 5 bytes message. It will be padded with 2 zero bytes and appended with a 1-byte length encoding. Hashing it will require only 1 field multiplication, instead of 2 field multiplication that the current scheme involves.

REFERENCES

[1]. Choosing a Good Hash Function, http://research.neustar.biz/2011/12/05/choosing-a-good-hash-function-part-1/ http://research.neustar.biz/tag/city-hash.

[2]. CityHash, https://code.google.com/p/cityhash.

[3]. R. Jenkins, http://burtleburtle.net/bob/c/frog.c.

[4]. SMHasher MurmurHash, https://code.google.com/p/smhasher.

[5]. S. Gueron and M. E. Kounavis. Intel Carry-Less Multiplication and Its Usage for Computing The GCM Mode, Rev 2.01. Intel Software Network. http://software.intel.com/sites/default/files/article/165685/clmul-wp-rev-2.01-2012-09-21.pdf.

[6]. S. Gueron and M. E. Kounavis. Efficient Implementation of the Galois Counter Mode Using a Carry-less Multiplier and a Fast Reduction Algorithm. Information Processing Letters 110: 549–-553 (2010).

APPENDIX

Listing 1. Software flow for reducing a 128-bit polynomial (string) modulo $P(x) = x^{64} + x^4 + x^3 + x + 1$. (note the assembly AT&T syntax, where the destination register is the rightmost operand) The flow uses two invocations of PCLMULQDQ. $P(x)$ is encoded as `poly = 0x1b`.

```
vpclmulqdq    $0x01, .Lpoly(%rip), ACC, T0      # reduction phase 1
vpand         .Land(%rip), ACC, ACC
vpxor         T0, ACC, ACC
vpclmulqdq    $0x01, .Lpoly(%rip), ACC, T0      # reduction phase  2
vpand         .Land(%rip), ACC, ACC
vpxor         T0, ACC, ACC
```

Listing 2. Software flow (C intrinsics) for computing S.

```
uint64_t UNIVERSAL_HASH_64_C(void *in, unsigned int len, uint64_t const_in){
__m128i ACC, DATA, T0;
__m128i KEY;
__m128i POLY = _mm_set_epi64x(0x00,0x1b);
__m128i ANDMASK = _mm_set_epi64x(0, 0xffffffffffffffff);
int len_save = len;
uint64_t rest = 0;
uint8_t *key_ptr = (uint8_t*)ks;
ACC = _mm_setzero_si128();

while(len>=16)
{
  DATA = _mm_loadu_si128(in);
  KEY = _mm_loadu_si128((__m128i*)key_ptr);
  ACC = _mm_xor_si128(ACC, _mm_clmulepi64_si128(DATA, KEY, 0x00));
  ACC = _mm_xor_si128(ACC, _mm_clmulepi64_si128(DATA, KEY, 0x11));
  in+=16;
  key_ptr+=16;
  len-=16;
}
if(len>=8)
{
  DATA = _mm_cvtsi64_si128(*(uint64_t*)in);
  KEY = _mm_cvtsi64_si128(*(uint64_t*)key_ptr);
  ACC = _mm_xor_si128(ACC, _mm_clmulepi64_si128(DATA, KEY, 0x00));
  in+=8;
  key_ptr+=8;
  len-=8;
}
if(len)
{
  uint8_t *r_ptr = (uint8_t*)&rest;
  while(len--)
  {
    *r_ptr++ = *(uint8_t*)in++;
  }
  DATA = _mm_cvtsi64_si128(rest);
  KEY = _mm_cvtsi64_si128(*(uint64_t*)key_ptr);
  ACC = _mm_xor_si128(ACC, _mm_clmulepi64_si128(DATA, KEY, 0x00));
  key_ptr+=8;
}
DATA = _mm_cvtsi64_si128(len_save<<3);
KEY = _mm_cvtsi64_si128(*(uint64_t*)key_ptr);
ACC = _mm_xor_si128(ACC, _mm_clmulepi64_si128(DATA, KEY, 0x00));
T0 = _mm_clmulepi64_si128(ACC, POLY, 0x01);
ACC = _mm_and_si128(ACC, ANDMASK);
ACC = _mm_xor_si128(ACC, T0);
T0 = _mm_clmulepi64_si128(ACC, POLY, 0x01);
ACC = _mm_and_si128(ACC, ANDMASK);
ACC = _mm_xor_si128(ACC, T0);
return _mm_cvtsi128_si64(ACC) ^ const_in;
}
```