IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 11, 2020, accepted February 5, 2020, date of publication February 20, 2020, date of current version February 28, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2974489

A Rolling Hash Algorithm and the Implementation to LZ4 Data Compression

HAO JIANG AND SIAN-JHENG LIN, (Member, IEEE)
CAS Key Laboratory of Electro-magnetic Space Information, School of Information Science and Technology, University of Science and Technology of China, Hefei 230027, China

Corresponding author: Sian-Jheng Lin (sjlin@ustc.edu.cn)

This work was supported in part by the Hundred Talents Program of Chinese Academy of Sciences, and Natural Science Foundation of Anhui Province (no. BJ2100330001).

ABSTRACT LZ77 is a dictionary compression algorithm by replacing the repeating sequence with the addresses of the previous referenced data in the stream. To find out these repetition, the LZ77 encoder maintains a hashing table, which have to frequently calculate hash values during the encoding process. In this paper, we present a class of rolling hash functions, that can calculate multiple hash values via a carry-less multiplication instruction. Then the proposed hash function is implemented in LZ4, which is a derivative of LZ77. The simulation shows that the encoding throughput of LZ4 has 15.7% improvement in average, and the compression ratio is $\pm 1\%$ in most cases.

INDEX TERMS Carry-less multiplication, LZ4, LZ77, rolling hash, SIMD.

I. INTRODUCTION

Data compression is a process of reducing data storage space, which is currently used in various aspects of software engineering. There are two major categories of compression algorithms, termed lossy and lossless [1]. The lossy compression algorithm reduces the size of a multimedia file, such as video, voice, and image, by removing small details that require a large amount of space [2]. Thus, it is impossible to restore the original file due to the removal of essential data. In contrast, the lossless compression is used in cases when the information must be completely restored [3]. The lossless data compression is used in text files, executable files, and source codes.

LZ77(Lempel-Ziv-1977) [4] is a class of lossless compression algorithms. LZ77 is a very simple adaptive dictionary-based technique, which does not require prior statistical characteristics of source [5]. Currently, there are many variants of LZ77 are proposed, such as LZ-Markov chain algorithm (LZMA) [6], LZ4 [7], LZB [8], LZP [9], and LZSS [10]. Although the implementations among them are slightly different, the objective of these algorithms is to find out the repeating sequences, which is usually achieved by hash functions and hash tables.

Hash function is a function mapping a block of data to a fixed-size code, called hash value. Hash functions can be used to detect repeating records in a large file. Nowadays, the hashing functions are widely used in many applications, such as secure encryptions, data deduplications, Bloom filters, and load balancing.

A good hash function satisfies two fundamental properties, termed simple calculation and uniform distribution. In particular, the simple calculation means that the computing time of the hash function should less than the time of other search and keyword comparison algorithms. The uniform distribution means that hash values are evenly distributed and the collisions are few.

When the encoder contains the components of the hash function, then these two properties of a hash function will affect compression respectively. The former property will affect the compression speed. And the latter one will affect the compression ratio, which we will be discussed in detail later.

A variety of fast hash functions have been proposed for requirements and applications. A rolling hash is used to prepare the calculation of piece hashes, It works by sampling the hash values of all substrings of a fixed length in a normalized string representation of its input [11]. An obvious application of rolling hash is used in Rabin-Karp string search algorithm [12], which is a substring searching algorithm. In data compression, LZ77 family uses the rolling hash to find out the repeating sequences in the data stream. The Bentley-McIlroy algorithm [13] uses a rolling hash to detect long repetitions that may occur far apart in the input text.

The associate editor coordinating the review of this manuscript and approving it for publication was Jun Wang.

In this paper, a rolling hash function for LZ4 is presented. However, the proposed hash function can also be applied to other variants of LZ77. In the proposed hash function, the input is treated as a binary polynomial $s(x)$, which is multiplied by a constant polynomial $p(x)$. Then the hash value is defined as the product with removing the high and low degree parts. An important property is that, this hash function can obtain multiple hash values by reading longer input sequence. In contrast, with other hash functions, the hash values should be calculate repeatedly. The contributions of this paper are enumerated as follows.

1) A hash function for rolling hash is proposed. In particular, the proposed hash function can calculate multiple hash values with using a carry-less multiplication instruction.
2) The proposed hash function is implemented in LZ4 library. The simulation shows that the encoding speed has 15.7% improvement in average, while the compression rate is basically identical.

The rest of this paper is organized as follows. In Section II, we review the rolling hash algorithms, the compression algorithms of LZ77 family, the hash functions used in LZ4 and the SIMD instruction set. Section III presents the proposed hash function. In Section IV, we show the details of implementing the proposed hash functions to LZ4 library. Section V gives the simulation of the proposed LZ4 algorithm. Section VI discusses a number of issues related to the proposed hash function. Section VII concludes this work.

II. RELATED WORKS

A. ROLLING HASH

The rolling hash is to sequentially calculate the hash values, which depends only on the substring in the sliding window. A number of rolling hash functions are proposed, and these algorithms maintains a state and each byte is added to the state as it is processed and removed from the state after a set number of other bytes have been processed [14].

The rolling hash function used in the Rabin-Karp string search algorithm is defined as

$$H(c_1, c_2, \ldots, c_k) = c_1 a^{k-1} + c_2 a^{k-2} + \ldots + c_k, \quad (1)$$

where a is a prime number, and the input integers $c_i, \ldots, c_k$ are the characters in the sliding window. Characters can be interpreted as integers with the coding system (e.g. ASCII, Unicode).

The next hash value can be calculated via

$$\begin{aligned} H(c_2, \ldots, c_{k+1}) &= c_2 a^{k-1} + c_3 a^{k-2} + \ldots + c_{k+1} \\ &= (H_{old} - c_1 a^{k-1})a + c_{k+1}, \end{aligned} \quad (2)$$

where $H_{old} = H(c_1, c_2, \ldots, c_k)$. Thus, the next hash value can be calculated with $O(1)$ operations by utilizing the previous hash value. This is the major difference from the conventional hash function, that requires $O(k)$ operations to calculate each hash value independently.

B. LZ77 FAMILY

LZ77 algorithms achieve compression by replacing the repeating sequence with the addresses of the previous referenced data in the stream. The algorithm searches the longest repetition of the current processing sequence in the sliding window. When a repetition is detected, it will be encoded as a pair of integers $\langle o, l \rangle$, where o is the offset, and l is the length of the repetition. If no repetition, the data is encoded as literals. Based on the specification of LZ77, it has better compression ratio for sequential data with many repetitions in context.

The process of finding repetitions requires sequence comparisons. The brute force way of comparing sequences is to compare the letters of two sequences, which has a time complexity $O(min(n_1, n_2))$, where n_1 and n_2 are the lengths of the two sequences. To accelerate the performance, the LZ77 implementation maintains a hash table to find out the repetitions. Precisely, the index of each sequence is saved in the hash table, and the sequence comparison requires $O(1)$ operation to calculate the hash function and query the index in the table.

In the hash table, different keywords may map to the same hash address. In this case, the LZ77 implementation is to replace the old entry with the new entry directly. This causes that the encoder may cannot find out longest repetitions. In addition, if the uniformity of a hash function is poor, most entries are concentrated in few buckets, and the collisions occur easily. This causes a lot of undetected repetitions, and the compression ratio tends to be unsatisfactory.

C. HASH FUNCTION IN LZ4

LZ4 is a byte-oriented compression scheme belonging to the LZ77 family focusing on compression and decompression speed [15], [16]. The hash function used in LZ4 is the multiply-shift hash [17], which is defined as

$$H(x) = \lfloor (ax \bmod 2^m)/2^{m-n} \rfloor, \quad (3)$$

where x is the input m-bit integer, a is a uniformly random odd m-bit integer. As illustrated above, the hash function (3) converts an m-bit integer to a n-bit integer. The standard LZ4 implementation chooses $(m, n) = (32, 13)$, and $a = 2654435761$ is a golden ratio prime.

Figure 1 gives a graphical representation. When the value ax is encoded as a binary representation, then the hash value $H(x)$ is a segment of ax between $m-n$ to $m-1$. When (3) is implemented in C with a 32-bit integer variable H, the code is given by

$$H = (a * x) \gg (m - n); \quad (4)$$

That is, the overflow discard in a 32-bit integer H is the same with the modulo 2^{32}, and the division with 2^{m-n} can be replaced with a right shift operation.

A standard hash function is the multiply-mod-prime scheme [18], which is defined as

$$H(x) = ((ax + b) \bmod p) \bmod n. \quad (5)$$

FIGURE 1. Multiply-shift hash, extracting bits $m-n, \ldots, m-1$ from the product ax as the hash value.

The article [18] reports that the hash function (3) is many times faster than the standard method (5).

D. ARITHMETIC IN $\mathbb{F}_2[x]$

The polynomial ring $\mathbb{F}_2[x]$ contains a set of binary single-variate polynomials. For each $a(x) \in \mathbb{F}_2[x]$, a binary polynomial $a(x) = \sum_{i=0}^{d_1} a_i x^i$, where d_1 is a non-negative integer and each $a_i \in \{0, 1\}$. The ring $\mathbb{F}_2[x]$ also defines two arithmetic operations, termed addition and multiplication, shown as follows.

Given two polynomials $a(x) = \sum_{i=0}^{d_1} a_i x^i \in \mathbb{F}_2[x]$ and $b(x) = \sum_{i=0}^{d_2} b_i x^i \in \mathbb{F}_2[x]$, the addition is defined as

$$a(x) + b(x) = \sum_{i=0}^{\max\{d_1,d_2\}} (a_i \oplus b_i) x^i,$$

where $\oplus$ is the exclusive OR (XOR) operation. In addition, the multiplication is defined as $a(x) \cdot b(x) = \sum_{i=0}^{d_1+d_2} c_i x^i$. Each symbol is given by

$$c_i = \sum_{j=\max\{0,i-d_2\}}^{\min\{i,d_1\}} a_j \odot b_{i-j}, \tag{6}$$

where $\odot$ is the AND operation, and $\sum$ is the summation modulo 2.

In implementations, we usually prefer to use binary representations to identify the polynomials in $\mathbb{F}_2[x]$. That is, the $a(x) = \sum_{i=0}^{d_1} a_i x^i \in \mathbb{F}_2[x]$ can be represented as an integer $a = (a_{d_1} a_{d_1-1} \ldots a_0)_2 = \sum_{i=0}^{d_1} a_i 2^i$. In this case, the addition is written as $a \oplus b$, where $\oplus$ is the bitwise XOR operation. The multiplication is written as $a \otimes b$, where $\otimes$ is the carry-less multiplication.

III. PROPOSED HASH FUNCTION

A. DEFINITION

The proposed hash function can be seen as the binary polynomial version of (3) in $\mathbb{F}_2[x]$. Precisely, given an input polynomial $s(x) = \sum_{i=0}^{d_2} s_i x^i$ with $d_2 < m$, we first calculate the product $r(x) = a(x) \cdot s(x) = \sum_{i=0}^{d_1+d_2} r_i x^i$, where $a(x) = \sum_{i=0}^{d_1} a_i x^i$, $d_1 = m - n$, is a constant polynomial. From (6), each coefficient of $r(x)$ is given by

$$r_i = \sum_{j=\max\{0,i-d_2\}}^{\min\{i,d_1\}} a_j s_{i-j}, \tag{7}$$

for $0 \le i \le d_1 + d_2$.

The hash value is a subset of the coefficients of $r(x)$. A good hash function has an important property that when even a bit of the input is altered, the hash value will change accordingly. Thus, we prefer that the chosen coefficients are related to all coefficients of $s(x)$. From (7), we have

$$r_i = \sum_{j=0}^{d_1} a_j s_{i-j}, \tag{8}$$

for $d_1 \le i \le d_2$. Thus, the hash value is defined as

$$r_{d_1} + r_{d_1+1} x + \cdots + r_{d_2} x^{d_2-d_1}.$$

From above, the proposed hash function is defined as follows.

Definition 1: Given an input polynomial $s(x) \in \mathbb{F}_2[x]$ with degree $\deg(s(x)) < m$, the proposed hash function is defined as

$$H(s(x)) = \lfloor (a(x) \cdot s(x) \bmod x^m) / x^{m-n} \rfloor, \tag{9}$$

where $a(x) \in \mathbb{F}_2[x]$ is a constant polynomial of degree $m-n$. In (9), the operation $\bmod x^m$ is to remove the terms $r_i x^i$, for $i \ge m$. In addition, the operation $/x^{m-n}$ is to remove the terms $r_i x^i$, for $i < m - n$. That is, the hash value is the coefficients between x^{m-n} and x^{m-1}.

Equivalently, if the input and output are treated as integers, the hash function (9) converts an m-bit integer to a n-bit integer. That is, (9) can be written as

$$H(s) = \lfloor (a \otimes s \bmod 2^m) / 2^{m-n} \rfloor, \tag{10}$$

where the integer $s < 2^m$ is the binary representation of $s(x)$, and the integer $a < 2^{m-n}$ is the binary representation of $a(x)$. For example, when $(m, n) = (5, 2)$, $s(x) = x^4 + x^3 + 1$ and $a(x) = x^3 + 1$, we have $r(x) = s(x) \cdot a(x) = x^7 + x^6 + x^4 + 1$, and $H(s(x)) = (r(x) \bmod x^5)/x^3 = x$. If the example is expressed as integers, then $s = (11001)_2 = 25$, $a = (1001)_2 = 9$, and $r = a \otimes s = (11010001)_2 = 321$. The hash value is $H(s) = (10)_2 = 2$.

B. MULTIPLE-HASH COMPUTATION

This subsection gives the scheme to calculate multiple hashes in a sliding window, when the hash function is Definition 1. Upon presenting the approach, we give a simple example as follows.

We consider $(m, n) = (3, 2)$, and $a(x) = x + 1$. The input sequence is denoted as (s_3, s_2, s_1, s_0). The size of the sliding window is three. To begin with, we take three symbols in the sliding window, and these symbols form a polynomial $s_0(x) = s_3 x^2 + s_2 x + s_1$. The hash value is given by

$$H(s_0(x)) = (s_3 \oplus s_2) x + (s_2 \oplus s_1). \tag{11}$$

Next, we move the sliding window a step, and obtain the polynomial $s_1(x) = s_2 x^2 + s_1 x + s_0$. The hash value is given by

$$H(s_1(x)) = (s_2 \oplus s_1) x + (s_1 \oplus s_0). \tag{12}$$

On the other hand, we construct a polynomial $s(x) = s_3 x^3 + s_2 x^2 + s_1 x + s_0$ by using all four symbols. Then we calculate

$$\begin{aligned} r(x) &= a(x)s(x) \\ &= s_3 x^4 + (s_3 \oplus s_2) x^3 + (s_2 \oplus s_1) x^2 + (s_1 \oplus s_0) x + s_0. \end{aligned} \tag{13}$$

From (11), (12) and (13), we have the following observations. First, there is a overlapping $s_2 \oplus s_1$ between $H(s_0(x))$ and

$H(s_1(x))$. Second, the degrees 2 and 3 of $r(x)$ are the coefficients of $H(s_0(x))$, and the degrees 1 and 2 of $r(x)$ are the coefficients of $H(s_1(x))$.

Therefore, instead of calculating $H(s_0(x))$ and $H(s_1(x))$ individually, we can calculate $r(x)$, and take a portion of $r(x)$ to obtain $H(s_0(x))$ (and $H(s_1(x))$, respectively). The following gives a formal theorem.

Theorem 1: Given a input polynomial

$$s(x) = s_{\ell-1}x^{\ell-1} + s_{\ell-2}x^{\ell-2} + \ldots + s_1 x + s_0 \in \mathbb{F}_2[x]$$

with degree $\deg(s(x)) < \ell$, a hash function is defined as

$$L_k(s(x)) = \lfloor (a(x) \cdot s(x) \bmod x^{\ell-k})/x^{\ell-k-n} \rfloor, \qquad (14)$$

where $0 \le k \le \ell - m$. Let

$$s_k(x) = s_{\ell-1-k}x^{m-1} + s_{\ell-2-k}x^{m-2} + \ldots + s_{\ell-m-k}.$$

Then $L_k(s(x)) = H(s_k(x))$.

Proof: Let $r(x) = a(x) \cdot s(x) = \sum_{i=0}^{\ell+m-n-1} r_i x^i$. From (14), we have $L_k(s(x)) = \sum_{i=1}^{n} r_{\ell-i-k}x^{n-i}$, where

$$r_{\ell-i-k} = \sum_{j=0}^{m-n} a_j s_{\ell-k-i-j}.$$

Thus, we can get

$$L_k(s(x)) = \sum_{i=1}^{n}(\sum_{j=0}^{m-n} a_j s_{\ell-k-i-j})x^{n-i}.$$

Let $r_k(x) = a(x) \cdot s_k(x) = \sum_{i=0}^{2m-n-1} r_{i,k}x^i$. From (9), we have

$$H(s_k(x)) = \sum_{i=1}^{n} r_{m-i,k}x^{n-i} = \sum_{i=1}^{n}(\sum_{j=0}^{m-n} a_j s_{\ell-k-i-j})x^{n-i}.$$

Thus, $L_k(s(x)) = H(s_k(x))$. ∎

Assume that the sliding window moves a byte forward after calculating a hash value. The new product will contain k hash values according to previous rules. Algorithm 1 gives the detail steps, where $k = (\ell - m)/8 + 1$ and $MASK = 2^n - 1$. Figure 2 gives an example for $(m, n) = (32, 13)$ and $\ell = 64$. It shows there is a overlap between any two adjacent hash values.

Algorithm 1 Multiple-hash computations

Input: A l-bit integer s
Output: k n-bit hash values

```
ret ← a ⊗ s
for i = 1, 2, ..., k do
    hi ← (ret ≫ (l − n − 8i + 8)) & MASK
end for
return (h1, h2, ··· , hk)
```

IV. IMPLEMENTATION TO LZ4

This section gives the details of implementing the proposed hash function to LZ4. First, the naive implementation is proposed. Then the algorithm is given to calculate multiple hash values in batch.

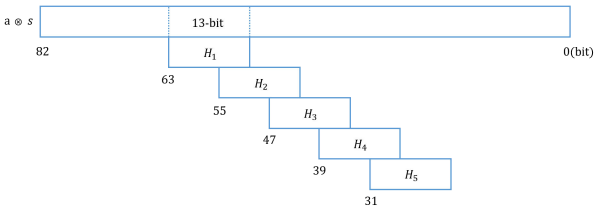

FIGURE 2. The implementation to calculate five hashes in LZ4, every 8 bits moved forward is the next hash value.

A. NAIVE IMPLEMENTATION

In this subsection, we present the naive implementation of the hash function in Definition 1. From the specification of LZ4, we use $(m, n) = (32, 13)$. That is, the input is an integer of $m = 32$ bits, and the constant a is chosen as an integer of $m-n+1 = 20$ bits. In this way, the product $a \otimes s$ has $2m-n = 51$ bits, and the middle $n = 13$ bits form the hash value. When the produce $a \otimes s$ is stored in a 32-bit integer variable, we do not need to perform the operation $\bmod x^m$ in (9), because the coefficients higher than degree 32 are overflow. The operation $/x^{m-n}$ can be implemented by the right shift operation $\gg$. Algorithm 2 gives the detail steps.

Algorithm 2 needs to perform the carry-less multiplication. Fortunately, the carry-less multiplication is implemented in certain SIMD instruction sets. For example, the instruction in ARMv8 is vmull_p64(), and the instruction in x86_64 is _mm_clmulepi64_si128(). In addition, when $a = 2^{m-n}+1$, the carry-less product $a \otimes s$ can be implemented with a left shift operation and a bitwise XOR operation. That is, Line 2 of Algorithm 2 can be replaced with

$$ret \leftarrow sx \oplus (sx \ll 19), \qquad (15)$$

which is usually faster than a carry-less multiplication on modern processors. Though the chosen $a = 2^{m-n} + 1$ may increase the probability of hash collisions, the simulation shows that the reduction of the compression ratio is limited.

Algorithm 2 Naive implementation to LZ4 on 32-bit CPU

Input: A 32-bit integer x
Output: A 13-bit hash value

```
1: ret ← a ⊗ x
2: ret ← ret ≫ 19
3: return ret
```

B. BATCH PROCESSING

Based on the approach in Section III-B, this subsection presents the implementation to calculate five hashes in LZ4. The implementation uses $(m, n) = (32, 13)$. That is, the encoder reads an integer of $\ell = 64$ bits, that is then multiplied with the constant a of $m - n + 1 = 20$ bits. Therefore, the product $a \otimes s$ has $\ell + m - n = 83$ bits, and we sequentially take five hash values from the product $a \otimes s$. The locations of these hash values are in [63, 51], [55, 43], [47, 35], [39, 27], [31, 19]. Clearly, we can also use the constant $a = 2^{m-n} + 1$ to improve the

Algorithm 3 Calculating five hash values in LZ4 on 64-bit CPU

Input: A 64-bit integer s

Output: Five 13-bit hash values

1: $ret \leftarrow a \otimes s$
2: $h1 \leftarrow (ret \gg 51)$ & $MASK$
3: $h2 \leftarrow (ret \gg 43)$ & $MASK$
4: $h3 \leftarrow (ret \gg 35)$ & $MASK$
5: $h4 \leftarrow (ret \gg 27)$ & $MASK$
6: $h5 \leftarrow (ret \gg 19)$ & $MASK$
7: **return** $(h1, h2, h3, h4, h5)$

TABLE 1. Configurations of simulation platforms.

Architecture	Core bit-width	CPU info	RAM
ARMv8-A	64	2.4GHz 64-core Cortex-A72	32GB
x86_64	64	4.2GHz 8-core i7-7700K	16GB

coding performance. Algorithm 3 gives the detail steps, and Figure 2 gives the graphical representation.

V. EXPERIMENTS

As the proposed hash function can only be applied to encoders, we only test the performance of the encoding in the experiment. The LZ4 v1.9.1 in its default mode (level 1) is chosen as the test program. We implement Algorithm 2 and Algorithm 3 in C, and the hash functions used in LZ4 are replaced with the proposed functions. All programs are compiled by GCC v7.4.0 with the optimization level -O3. All experiments are performed by a single thread on the platforms with ARMv8 architecture processors and x86_64 architecture processors, respectively. Table 1 tabulates the configurations of the platforms.

The data sets used in the experiments are chosen from the Calgary corpus [19] and the Canterbury corpus [20]. The first experiment shows the compression ratios of LZ4 with various hash functions, where the column Con. is the compression ratio of the conventional LZ4. Two constants are tested, namely $a_0 = 2^{19} + 2^6 + 2^2 + 2^1 + 1$ and $a_1 = 2^{19} + 1$. Table 2 lists the compression ratios for 13 input files. As shown in Table 2, the compression ratios of the proposed hash functions are similar to that of the conventional LZ4. For the hash function with a_0, the compression ratio of the proposed approach is about 0.0023% worse than the conventional hash function in average. For the hash function with a_1, the compression ratio of the proposed approach is about 0.554% worse than the conventional hash function in average. This shows that the uniformity of the proposed hash function is close to that of the hash function in LZ4.

In the second experiment, we consider the throughput of the encoders. The throughput is defined as the amount of data read per second (MB/sec). Table 3 lists the throughput of the programs on the platforms with ARMv8 processors and x86_64 processors, respectively. In addition, we list the ratio between batch method and conventional LZ4, and that

TABLE 2. Compression ratios of LZ4 with various hashing functions.

File name	Category	Con.	a_0	a_1
progp	Source code in PASCAL	37.90	37.55	37.54
progc	Source code in "C"	52.77	52.88	53.27
obj1	Object code for VAX	60.11	60.14	60.85
paper1	Technical paper	54.43	54.54	54.89
paper3	Technical paper	60.81	60.74	61.52
paper4	Technical paper	63.76	63.47	65.42
paper5	Technical paper	62.37	62.34	62.93
paper6	Technical paper	54.08	54.70	54.92
cp.html	HTML source	48.39	48.31	48.75
sum	SPARC Executable	49.19	49.05	50.47
xargs.1	GNU manual page	62.88	62.67	62.83
grammar.lsp	LISP source	51.38	51.57	51.25
fields.c	C source	46.77	46.91	47.41

is defined as

$$Ratio = \frac{\text{Ours\_Batch - Conventional}}{\text{Conventional}}.$$

Though the carry-less multiplication is supported by the SIMD instruction set, the throughput of our naive implementation is around 77.65% (and 76.36%) of that of the conventional LZ4 on x86_64 processors (and on ARMv8 processors). This is because the number of cycles for the carry-less multiplication instruction is greater than the number of cycles for the integer multiplication on modern CPUs.

For our batch implementations, the proposed algorithm usually has higher throughput than the conventional LZ4 on x86_64 processors. It shows that the propose algorithm has about $Ratio^{avg}_{x86\_64} = 12.44\%$ improvements on x86_64 processors in average. Further, the proposed algorithm always has higher throughput than the conventional LZ4 on ARMv8 processors. It shows that the propose algorithm has about $Ratio^{avg}_{ARMv8} = 18.89\%$ improvements on ARMv8 processors in average.

VI. DISCUSSIONS

A. PERFORMANCE OF THE PROPOSED HASH FUNCTION

Table 3 shows that the proposed batch implementation can improve the encoding throughput. In Algorithm 3, the encoder calculates five hash values via (15), that requires a left shift operation and a bitwise XOR operation. Then in the next four hashing rounds, the encoder does not need to call the hash function again, and the hash value is in the set of prior obtained values. In contrast, the conventional hash function (4) in LZ4 requires an integer multiplication, that takes more cycles than the bitwise operations used in the proposed implementation. Thus, the proposed hash function is faster than the the conventional hash function (4) in LZ4.

As shown in Table 3, the improvement Ratio is variant. A reason is if the LZ4 encoder finds out a repetition in the sequence, the encoder will almost skip the hash computations for the repeating sequence. Thus, when the input sequence has many repeating sequences, most hash values calculated by Algorithm 3 may be discarded (except for the first one). This causes that the throughput of the proposed implementation is close to that of the conventional LZ4.

TABLE 3. Encoding throughput (MB/sec) of the LZ4 and the modified LZ4 on two platforms.

File Name	x86_64				ARMv8			
	Conventional	Ours_Naive	Ours_Batch	*Ratio*	Conventional	Ours_Naive	Ours_Batch	*Ratio*
progp	825.2	616.3	843.1	2.17%	331.7	272.4	360	8.53%
progc	565	428.4	590.3	4.48%	235.4	185.7	256.2	8.84%
obj1	929.5	671	1258.4	35.38%	254.7	181.1	350.9	37.77%
paper1	486	370.4	485.9	-0.02%	216.9	176.8	232.3	7.10%
paper3	446.9	338.2	452.1	1.16%	191.1	163	210.9	10.36%
paper4	655.4	554.3	909.4	38.75%	201.3	162.2	261.2	29.76%
paper5	717.8	590.7	905.7	26.18%	206.5	169	258.9	25.38%
paper6	549.7	406.4	562.7	2.36%	225.3	178.4	250.7	11.27%
cp.html	814.3	653.4	1004	23.30%	261.7	198.5	308.3	17.81%
sum	725.4	582	809.5	11.59%	269	204.1	309.2	14.94%
xargs.1	852.1	622.9	898.3	5.42%	234.5	174	313.2	33.56%
grammar.lsp	991.9	722	1033.8	4.22%	296.5	214.6	374.2	26.21%
fields.c	986.4	732.9	1053.2	6.77%	329.1	246.8	375.4	14.07%

B. DRAWBACKS OF THE CONVENTIONAL ROLLING HASH

In this paper, we do not adopt the conventional rolling hash method (see Section II-A). The major reason is that, calculating (2) requires two multiplications and two additions. In contrast, the hash function (4) used in LZ4 only requires a multiplication and a right shift operation. Thus, we conclude that calculating (2) is slower than calculating (4) on modern processors.

VII. CONCLUSION

In this paper, we present a new hash function, that is suitable for the application that need to calculate hash values in a sliding window. In particular, the proposed hash function can calculate multiple hash values with a carry-less multiplication instruction. The proposed hash function is implemented on the LZ4 compression algorithm. The experiments show that the proposed hash function has similar compression ratio to the conventional hash function in LZ4. Thus, it meets the requirements in actual applications. Moreover, the proposed algorithm generally improves the encoding throughput on x86_64 and ARMv8 processors.

REFERENCES

[1] K. Sayood, *Introduction to Data Compression*. San Mateo, CA, USA: Morgan Kaufmann, 2017.
[2] G. K. Wallace, "The JPEG still picture compression standard," *IEEE Trans. Consum. Electron.*, vol. 38, no. 1, pp. 18–34, 1992.
[3] K. Kim, C. Lee, and H.-J. Lee, "A sub-pixel gradient compression algorithm for text image display on a smart device," *IEEE Trans. Consum. Electron.*, vol. 64, no. 2, pp. 231–239, May 2018.
[4] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.
[5] W. Liu, F. Mei, C. Wang, M. O'Neill, and E. E. Swartzlander, "Data compression device based on modified LZ4 algorithm," *IEEE Trans. Consum. Electron.*, vol. 64, no. 1, pp. 110–117, Feb. 2018.
[6] B. Li, L. Zhang, Z. Shang, and Q. Dong, "Implementation of LZMA compression algorithm on FPGA," *Electron. Lett.*, vol. 50, no. 21, pp. 1522–1524, Oct. 2014.
[7] Y. Collet. *LZ4—Extremely Fast Compression*. Accessed Dec. 13, 2019. [Online]. Available: https://lz4.github.io/lz4/
[8] T. Bell, "A unifying theory and improvements for existing approaches to text compression," Ph.D. dissertation, Dept. Comput. Sci., Univ. Canterbury, Christchurch, New Zealand, 1986.
[9] C. Bloom, "LZP: A new data compression algorithm," in *Proc. Data Compress. Conf. (DCC)*, Dec. 2002, p. 425.
[10] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, Oct. 1982.
[11] M. J. Thomsen and F. Henglein, "Clone detection using rolling hashing, suffix trees and dagification: A case study," in *Proc. 6th Int. Workshop Softw. Clones (IWSC)*, Jun. 2012, pp. 22–28.
[12] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Develop.*, vol. 31, no. 2, pp. 249–260, Mar. 1987.
[13] J. Bentley and D. McIlroy, "Data compression using long common strings," in *Proc. DCC Data Compress. Conf.* , 1999, pp. 287–295.
[14] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digit. Invest.*, vol. 3, pp. 91–97, Sep. 2006.
[15] D. Harnik, E. Khaitzin, D. Sotnikov, and S. Taharlev, "A fast implementation of deflate," in *Proc. Data Compress. Conf.*, Mar. 2014, pp. 223–232.
[16] M. Bartik, S. Ubik, and P. Kubalik, "LZ4 compression algorithm on FPGA," in *Proc. IEEE Int. Conf. Electron., Circuits, Syst. (ICECS)*, Dec. 2015, pp. 179–182.
[17] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, "A reliable randomized algorithm for the closest-pair problem," *J. Algorithms*, vol. 25, no. 1, pp. 19–51, Oct. 1997.
[18] M. Thorup, "High speed hashing for integers and strings," 2015, *arXiv:1504.06804*. [Online]. Available: http://arxiv.org/abs/1504.06804
[19] *Calgary Corpus*. Accessed: Dec. 13, 2019. [Online]. Available: http://www.data-compression.info/Corpora/CalgaryCorpus/
[20] *Canterbury Corpus*. Accessed: Dec. 13, 2019. [Online]. Available: http://www.data-compression.info/Corpora/CanterburyCorpus/

HAO JIANG received the B.E. degree in radio and television engineering from the Communication University of Zhejiang (CUZ), Hangzhou, China, in 2018, where he is currently pursuing the M.Sc. degree. His research focuses on the data compression.

SIAN-JHENG LIN (Member, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees in computer science from National Chiao Tung University, Hsinchu, Taiwan, in 2004, 2006, and 2010, respectively. From 2010 to 2014, he was a Postdoctoral Researcher with the Research Center for Information Technology Innovation, Academia Sinica. From 2014 to 2016, he was a Postdoctoral Researcher with the Electrial Engineering Department, King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia. He was a part-time Lecturer with Yuanpei University, from 2007 to 2008, and Hsuan Chuang University, from 2008 to 2010. He is currently a Project Researcher with the School of Information Science and Technology, University of Science and Technology of China (USTC), Hefei, China. In recent years, his research focuses on the algorithms for MDS codes and its applications to storage systems.