Construct a Perfect Word Hash Function in Time Independent of the Size of Integers

Yijie Han[1]

School of Computing and Engineering, University of Missouri at Kansas City, Kansas City, MO 64110, USA
hanyij@umkc.edu

Abstract. We present an algorithm for constructing a perfect word hash function for n integers that takes $O(n^4 \log n)$ time. This time is independent of size of the integers or the number of bits in the integers. We call it a word hash function because we require that the hash function can hash multiple integers packed in a word in constant time. Previous algorithms for constructing a perfect word hash function have time dependent on the number of the bits in integers. Although an $O(n(\log\log n)^2)$ time algorithm is known for constructing a perfect hash function, the hash function constructed is not a word hash function and it cannot hash multiple integers packed in one word in constant time. The property of word hashing is indispensable in the current best deterministic and randomized algorithms for integer sorting. Our result is achieved via an algorithm of $O(n^2 \log^2 n)$ time that computes the shift distances for integers of $\Omega(n^2 \log n)$ bits. These shift distances can then be used to pack the extracted bits of each integer to $O(n)$ bits. Perfect word hash function constructed with our method using these shift distances allows a batch of mn integers with m integers packed in a word to be hashed in $O(n)$ time.

Keywords: Analysis of algorithms, design of algorithms, hashing, perfect word hash functions, integers.

1 Introduction

A hash function is a function mapping integers in $\{0, 1, ..., m-1\}$ to integers in $\{0, 1, ..., m'-1\}$ with $m' < m$. When applying a hash function to n integers two integers may be mapped to the same value. This is called a collision. A perfect hash function on n integers is a hash function that has no collision for these n integers. Perfect hash functions have been studied by many researchers [2, 5–8, 13–15]. A word hash function is a hash function that can hash multiple integers packed in a word in constant time. Previous known perfect word hash functions require construction time dependent on the number of bits of integers to be hashed. Thus when dealing with very large integers these perfect hash functions are at disadvantage as when we are constructing a perfect hash function for n integers the time for construction cannot be bounded by a polynomial of n. Earlier Fredman et al. provided a perfect hash function [6] which requires $O(n^3 \log m)$ time to construct, where $\log m$ is the number of bits in an input integer (i.e. integers to be hashed are taken from $\{0, 1, ..., m-1\}$). Dietzfelbinger et al. gave a randomized word hash function in [5] and Raman showed [14] how to derandomize it to obtain a deterministic perfect word hash function in time $O(n^2 \log m)$. Thus the construction time of these hash functions depends on the number of bits of integers and their time cannot be written as a polynomial of n. Ruzic [16] presented an algorithm with $O(n(\log\log n)^2)$ time to construct a perfect hash function. Thorup published an algorithm [17] that builds a dictionary for integers such that the access time is $O(\log n/\log w)$, where w is the word length. Hagerup et al. [8] obtained a perfect hash function with construction time $O(n \log n)$. However, Ruzic's algorithm [15], Pătraşcu and Thorup's algorithm [17], and the algorithm of Hagerup et al. [8] can only be applied to individual integers and they cannot be applied to hash multiple integers packed in one word in constant time. That is: they are not word hashing algorithms.

In this paper we give an algorithm h_1 for hashing n integers of $\Omega(n^2 \log n)$ bits to $O(n)$ bits integers in $O(n)$ time. h_1 is obtained after we computed shift distances $d_1, d_2, d_3, ...$ (to be explained in the following sections). These shift distances can be computed in $O(n^2 \log^2 n)$ time for integers of $\Omega(n^2 \log n)$ bits. Thus for integers of $\Omega(n^2 \log n)$ bits we can construct a perfect hash function $r_1 \circ h_1$ in $O(n^2 \log^2 n + n^2 \log m) = O(n^3)$ time (where $\log m = n$), where r_1 is Raman's hash algorithm [14] (originally defined in [5]) with hash function construction time $O(n^2 \log m)$. For integers of $O(n^2 \log n)$ bits we can use Raman's algorithm [14] directly to construct a perfect hash function in $O(n^2 \log m) = O(n^4 \log n)$ time.

Thus overall we can construct a perfect hash function in $O(n^4 \log n)$ time. In our method hashing has to be done in batches of n integers and the hash time for mn integers with m integers packed in a word is $O(n)$.

We can reduce the time for computing the shift distances to $O(n^2 \log n)$. Then the hashing of n words would take $O(n \log n)$ time instead of $O(n)$ time in this case.

A very important feature we emphasizing is that our hash function is a word hashing function. That is, it allows multiple integers packed in one word to be hashed in constant time. This word hashing property is indispensable in the design of fast integer sorting algorithms. Current best deterministic algorithms [9][10] and randomized algorithms [1][3][12] for integer sorting all require the use of word hashing. Our recent result on ordered partition [11] also uses word hashing. Without using word hashing these algorithms cannot achieve their claimed time bounds.

This word hashing property was exhibited in the design of a randomized hash function of Dietafelbinger et al. [5]. Raman derandomized Dietzfelbinger's algorithm [14] and thus the deterministic hash algorithm is a word hash algorithm. However Raman's derandomization takes $O(n^2 w)$ time with w bits integers and thus is not strictly a polynomial time algorithm. Ruzic's algorithm [16], Pătraşcu and Thorup's algorithm [17], and the algorithm of Hagerup et al. [8] are not word hashing algorithms.

The computation model we used for our algorithms is the RAM model [4]. We assume unit cost for $+$, $-$, $*$, $/$, $\vee$, $\wedge$, xor, not, shift. In particular logarithm function is not charged at unit cost level.

The perfect hash function described here finds application in linear time deterministic ordered partition [11], i.e., partition n integers to $\sqrt{n}$ sets S_i, $0 \leq i < \sqrt{n}$, where $|S_i| = O(\sqrt{n})$ and $S_0 \leq S_1 \leq \cdots \leq S_{\sqrt{n}-1}$ ($S_i \leq S_{i+1}$ means $\max S_i \leq \min S_{i+1}$).

Current integer sorting can be done in $O(n \log \log n)$ deterministic time and linear space[9]. The algorithms presented in this paper may help in the search of an optimal algorithm for integer sorting.

The Overall Approach

There are two components in our approach, the first component is to construct the perfect hash function which we show how to do it in $O(n^4 \log n)$ time. This construction is conditioned on the number of bits of each input integer. If each integer has $O(n^2 \log n)$ bits then we use Raman's algorithm [14] with time $O(n^2 w) = O(n^4 \log n)$ for constructing the perfect hash function h_1, where w is the number of bits in each integer, i.e. $O(n^2 \log n)$. If each integer has $\Omega(n^2 \log n)$ bits, then we extract $n-1$ bits for each integer and these $n-1$ bits can be used to identify this integer, i.e. $a \neq b$ iff the extracted $n-1$ bits of a and b are not equal. The extraction of these $n-1$ bits from every integer can be done in $O(n)$ time provided some shifting distances are known. The computing and finding these shifting distances is part of the construction of "the perfect hash function" and it has $O(n^2 \log^2 n)$ time. After each integer is reduced to $n-1$ bits then Raman's algorithm is again used for constructing the perfect hash function h_2. The second component is the hashing. h_1 and h_2 are actually odd integers a_1 representing h_1 and a_2 representing h_2. According to Dietzfelbinger et al. [5] to hash integer x with h_1 (h_2) is to compute $h_1(x) = (a_1 x \bmod 2^{k_1}) \text{ div } 2^{k_1 - 2\log n}$ ($h_2(x) = (a_2 x \bmod 2^{k_2}) \text{ div } 2^{k_2 - 2\log n}$). Where 2^{k_1} (2^{k_2}) is the range of the integers to be hashed, i.e. $O(2^{n^2 \log n})$ ($O(2^n)$). $2^{2\log n}$ is the range of the hashed integers (set in Raman's algorithm [14]). h_1 and h_2 are word hash functions (mod and div can be done by AND and shift). Thus when integer has $O(n^2 \log n)$ bits we use h_1 to hash and when integer has $\Omega(n^2 \log n)$ bits we use $O(n)$ time to convert them to $n-1$ bits and then use h_2 to hash. Because the second component is straightforward we mainly describe the first component in our paper.

2 Extracting Bits

To construct a perfect hash function for n integers we will first sort these n integers. Let $a_0 < a_1 < a_2 < \cdots < a_{n-1}$ be the sorted integers (all integers with the same value can be excluded except one). Let $msb(a)$ be the index of the most significant bit of a that is 1, where index counts starting from least significant bit at 0. We compute $m(i) = msb(a_i \oplus a_{i+1})$, $0 \leq i < n-1$, where $\oplus$ is the bit-wise exclusive-or operation. We take all the bits indexed in $M = \{m(i) \mid i = 0, 1, ..., n-2\}$ for each integer a_j to form a'_j, $0 \leq j < n$. Thus now each a'_j has at most $n-1$ bits. $a_i < a_j$ iff $a'_i < a'_j$.

Example 1: Let $a_0 = 0100001, a_1 = 0100101, a_2 = 0110000$, then the most significant bit a_0 and a_1 differ is the 2nd bit (counting from the least significant bit) and the most significant bit a_1 and a_2 differ is the 4th bit, thus $M = \{2, 4\}$ and $a'_0 = 00, a'_1 = 01, a'_2 = 10$ (the 2nd and the 4th bits). □

There are two problems here. The first problem is how to obtain set M. The second problem is after bits are extracted how do we pack them to $n-1$ consecutive bits.

First we will not compute M but instead compute $M' = \{2^{msb(a_i \oplus a_{i+1})} \mid i = 0, 1, ..., n-2\}$. In fact even M' is difficult to compute and we will adapt our method.

The second problem will be solved in the following sections.

As we said that M' is difficult to compute. We will instead use the least significant bit. Let $lsb(a)$ be the index of the least significant bit of a that is 1. Note that it will be easy to extract the least significant bit that is 1. To extract the least significant bit of a that is 1 simply do $2^{lsb(a)} = (a \oplus (a-1)) + 1)/2$. Next we will *view* each integer reverse-wards, that is, we view the least significant bit as the most significant bit and the most significant bit as the least significant bit. As will be shown that we will use this order to sort the n integers. We will call this order as the least significant bit order. Now the approach we described earlier will work if we sorted integers by the least significant bit order, i.e. say $a''_0, a''_1, ..., a''_{n-1}$ are the integers sorted by the least significant bit order, then the least significant bit that a''_i and a''_{i+1} differ, $i = 0, 1, ..., n-2$, will give us $n-1$ bits that make integers differ between each other. We let $m'(i) = lsb(a''_i \oplus a''_{i+1})$. There are at most $n-1$ different values for $m'(i)$, $0 \leq i < n-1$. Now to sort integers by the least significant bit order we use comparison sorting and compare integers a and b by examining $(2^{lsb(a \oplus b)} \vee a) == a$ and $(2^{lsb(a \oplus b)} \vee b) == b$, where $\vee$ is the bit-wise OR operation. If $(2^{lsb(a \oplus b)} \vee a)$ is equal to a then a is "larger" than b in the least significant bit order.

After we get $a''_0, a''_1, ..., a''_{n-1}$ we then compute $L_i = 2^{m'(i)} = 2^{lsb(a''_i \oplus a''_{i+1})}$, $i = 0, 1, ..., n-2$, to get the least significant bits. Let $L = \vee_{i=0}^{n-2} L_i$. L provides the mask for us to extract out needed bits as we now do $b_i = a_i \wedge L$, $i = 0, 1, ..., n-1$, where $\wedge$ is the bit-wise AND operation. Note that no more than $n-1$ bits will be extracted from each integer.

3 Pack Bits

In the last section we showed how to extract at most $n-1$ bits from each integer. These extracted bits need to be packed. In this section we will show how to pack into $O(n)$ bits with integers of $\Omega(n^4)$ bits. We will compute shift distances $d_1, d_2, ...,$. These shift distances can be computed in $O(n^4)$ time. In the later sections we show how to improve the algorithm to work with integers of $\Omega(n^2 \log n)$ bits.

Without loss of generality we assume that all L_i's are different.

Note that in Fredman and Willard [7] Lemma 2 it was shown that these extracted $n-1$ bits (scattered bits of 1's among $\log m$ bits) can be packed to n^4 bits by multiplying a multiplier and this multiplier can be computed in $O(n^4)$ time. However, the method in [7] requires that set $M_2 = \{m'(i) \mid i = 0, 1, ..., n-1\}$ be obtained. In the last section we only obtained $M_1 = \{2^{m'(i)} \mid i = 0, 1, ..., n-1\}$. To obtain M_2 from M_1 we need to apply a logarithm which may not be readily available. Fredman and Willard's method do have the advantage of hashing one integer at a time. Our method requires the hashing of batches of n integers at a time.

Because there are only $n-1$ extracted bits there are less than n^2 different distances (the set of distances is $D = \{|m'(i) - m'(j)|, 0 \leq i, j \leq n-2\}$) among them. By trying out n^2 different distances $n+1, n+2, ..., n+n^2$ (the reason we have this additive n is because we want to shift at least n bits to save enough space for packing later) we will find a distance not in D. Because we did not obtain M_2 we check a distance d_1, $d_1 = n+1, n+2, ..., n+n^2$, by trying out $(L \vee (L \rightarrow d_1))) == (L + (L \rightarrow d_1))$, where $\rightarrow d_1$ is shift d_1 bits to the right. If it is equal then distance d_1 is available (not in D). Let $b_i = a_i \wedge L$ contain the extracted bits. After we find an available distance d_1 we do $b_{i/2} = b_i \vee (b_{i+1} \rightarrow d_1)$,

$i = 0, 2, 4, ...$ We also do $L'_0 = L_0 \leftarrow d_1$, $L'_1 = L_0$, $L_i = L_i \vee (L_i \rightarrow d_1)$, $i = 0, 1, ..., n-2$, and $L = L \vee (L \rightarrow d_1)$. L'_i's indicate the location of $m'(0)$'s and they will be used later to extract the packed bits. Now we have $n/2$ integers and each integer has $2(n-1)$ extracted bits. Therefore there are no more than $4n^2$ distances among them. We pick an available distance d_2 among $n+1, n+2, ..., n+4n^2$ using the same method and then do $b_{i/2} = b_i \vee (b_{i+1} \rightarrow d_2)$, $i = 0, 2, 4, ...$ Also $L'_2 = L'_0$, $L'_0 = L'_0 \leftarrow d_2$, $L'_3 = L'_1$, $L'_1 = L'_1 \leftarrow d_2$, $L_i = L_i \vee (L_i \rightarrow d_2)$, $i = 0, 1, ..., n-2$, and $L = L \vee (L \rightarrow d_2)$. After we do this $\log n$ times we have all the $n(n-1)$ extracted bits in all n integers $\vee$-ed into one integer b_0. The time spent is $O(n^4)$. We then extract the $m'(i)$-th bits by doing $m_i = b_0 \wedge L_i$. Note that m_i contains the $m'(i)$-th bits of all a_j, $j = 0, 1, ..., n-1$. Note also that the bit patterns in L_i and L_j are exactly the same (i.e. L_j can be obtained from L_i by shifting L_i by $\log(L_j/L_i)$ bits (because logarithmic function is not readily available we can substitute shifting $\log(L_j/L_i)$ bits by multiplying L_j/L_i)). Now we pack integers together by doing

for($i = 1$; $i <= n-2$; $i++$)
{
    $m_0 = m_0 \vee (m_i * L_0/(2^i L_i))$;
}

Because we added an addend n when we do shifting previously to save space and therefore there will be enough space when we pack integers here (i.e. integers will not be mixed up). After integers are packed we can then obtain packed integer b'_i (packed from b_i) as $L''_i = (L'_i \leftarrow 1) - (L'_i \rightarrow (n-1))$ (obtain mask and $\leftarrow 1$ is shift left by 1 bit) and $b'_i = m_0 \wedge L''_i$. Now to move extracted bits to the least signigicant $n-1$ bits do $b'_i = b'_i/(L'_i \rightarrow (n-1))$.

Example 2: Let $b_0 = 000a0a00000aa0$, $b_1 = 000b0b00000bb0$, $b_2 = 000c0c00000cc0$, $b_3 = 000d0d00000dd0$, where a, b, c, d are extracted bits. $L_0 = 00010000000000$, $L_1 = 00000100000000$, $L_2 = 00000000000100$, $L_3 = 00000000000010$, $L = 00010100000110$.

Take $d_1 = 5$. Then do $b_0 = b_0 \vee (b_1 \rightarrow 5) = 000a0a00b0baa000bb0$, $b_1 = b_2 \vee (b_3 \rightarrow 5) = 000c0c00d0dcc000dd0$, $L'_0 = L_0 \leftarrow 5 = 0001000000000000000$, $L'_1 = L_0 = 0000000010000000000$ (thus L'_0 indicates the position of first a or c and L'_1 indicates the position of first b or d). $L_0 = L_0 \vee (L_0 \rightarrow 5) = 0001000010000000000$, $L_1 = L_1 \vee (L_1 \rightarrow 5) = 0000010000100000000$, $L_2 = L_2 \vee (L_2 \rightarrow 5) = 0000000000010000100$, $L_3 = L_3 \vee (L_3 \rightarrow 5) = 0000000000001000010$, $L = L \vee (L \rightarrow 5) = 0001010010111000110$.

Take $d_2 = 15$. Then do $b_0 = b_0 \vee (b_1 \rightarrow 15) = 000a0a00b0baa000bbc0c00d0dcc000dd0$, $L'_2 = L'_0 =$ 0000000000000000001000000000000000,
$L'_0 = L'_0 \leftarrow 15 = 0001000000000000000000000000000000$,
$L'_3 = L'_1 = 0000000000000000000000010000000000$,
$L'_1 = L'_1 \leftarrow 15 = 0000000010000000000000000000000000$ (thus L'_0 indicates the position of first a, L'_1 indicates the position of first b, L'_2 indicates the position of first c, L'_3 indicates the position of first d.) And

$L_0 = L_0 \vee (L_0 \rightarrow 15) = 0001000010000000001000010000000000,$
$L_1 = L_1 \vee (L_1 \rightarrow 15) = 0000010000100000000010000100000000,$
$L_2 = L_2 \vee (L_2 \rightarrow 15) = 0000000000010000100000000010000100,$
$L_3 = L_3 \vee (L_3 \rightarrow 15) = 0000000000001000010000000001000010.$
$L = L \vee (L \rightarrow 15) = 0001010010111000111010010111000110.$

Here we see that L_0, L_1, L_2, L_3 have the same pattern and they differ by only a shift of bits.

Now compute

$m_0 = b_0 \wedge L_0 = 000a0a00b0baa000bbc0c00d0dcc000dd0 \wedge$
$0001000010000000001000010000000000 = 000a0000b000000000c0000d0000000000,$

$m_1 = b_0 \wedge L_1 = 000a0a00b0baa000bbc0c00d0dcc000dd0 \wedge$
$0000010000100000000010000100000000 = 00000a0000b000000000c0000d00000000,$

$m_2 = b_0 \wedge L_2 = 000a0a00b0baa000bbc0c00d0dcc000dd0 \wedge$
$0000000000010000100000000010000100 = 00000000000a0000b000000000c0000d00,$

$m_3 = b_0 \wedge L_3 = 000a0a00b0baa000bbc0c00d0dcc000dd0 \wedge$
$0000000000001000010000000001000010 = 000000000000a0000b000000000c0000d0.$

Now do

$m_0 = m_0 \vee (m_1 * L_0/(2L_1)) = m_0 \vee (m_1 * 2);$
$m_0 = m_0 \vee (m_2 * L_0/(4L_2)) = m_0 \vee (m_2 * 2^6);$
$m_0 = m_0 \vee (m_3 * L_0/(8L_3)) = m_0 \vee (m_3 * 2^6);$

We get that $m_0 = 000aaaa0bbbb000000cccc0dddd0000000.$

That is, bits for each integer are packed together.

Now do

$L_0'' = (L_0' \leftarrow 1) - (L_0' \rightarrow 3) = 0010000000000000000000000000000000-$
$0000001000000000000000000000000000 = 0001111000000000000000000000000000$ (obtain mask)

and

$b_0' = m_0 \wedge L_0'' = 000aaaa0bbbb000000cccc0dddd0000000 \wedge$
$0001111000000000000000000000000000 = 000aaaa000000000000000000000000000$

Now do

$b'_0 = b'_0/(L'_0 \to 3) = aaaa$.

Similarly *bbbb* for b'_1, *cccc* for b'_2, *dddd* for b'_3 can also be extracted. □

Theorem 1: For integers of $\Omega(n^4)$ bits a perfect hash function can be constructed in $O(n^4)$ time for hashing n integers to $O(n)$ bits integers (pack the $n-1$ bits with indices in $\{m'(0), m'(1), ..., m'(n-1)\}$ in each integer to $n-1$ bits in $O(n^4)$ time). Thereafter the hashing of a batch of n integers to $O(n)$ bits integers takes $O(n)$ time. □

Note that the time $O(n^4)$ in Theorem 1 is actually the time for computing shift distances $d_1, d_2, ...$ Thus this time affects the time for constructing a perfect hash function as a perfect hash function can be constructed by first packing the extracted bits to $n-1$ bits and then use Raman's algorithm [14] to construct the perfect hash function in $O(n^2 \log m) = O(n^3)$ time. After shift distanced $d_1, d_2, ...$ have been computed we can then pack the extracted bits for a batch of n integers in $O(n)$ time by the algorithm presented in this section. The only requirement for packing is that integers have $\Omega(n^4)$ bits. Thus for integers of $O(n^4)$ bits the construction time for a perfect hash function in Raman's algorithm becomes $O(n^2 \log m) = O(n^6)$.

4 Construction in $O(n^2 \log n)$ Time

In this section we show the improvement to pack the extracted bits to $O(n)$ bits in $O(n^2 \log n)$ time, i.e. to compute the shift distances in $O(n^2 \log n)$ time.

First note that the time $O(n^4)$ for packing in the last section can really be reduced to $O(n^3)$ time by a better analysis. Note that after we $\vee$-ed integers $b_0, b_1, ..., b_{2^i-1}$ into one integer b the number of possible distances in b is much smaller than $(2^i(n-1))^2$. This is because the distance in b between two bits can be represended as $\pm k + \delta_1 d_1 + \delta_2 d_2 + \cdots + \delta_i d_i$, where k is a distance taking from the $(n-1)^2$ distances in a b_j and each δ_j can assume the value of 0 or 1. Thus the number of possible distances is bounded by $2(n-1)^2 \cdot 2^i$. This analysis will bound in the the number of possible distances to $O(n^3)$ and therefore the time complexity of the algorithm in Theorem 1 can be reduced to $O(n^3)$.

In last section we first $\vee$-ed all bits into b_0. This requires $O(n^4)$ bits as there are a total of $n(n-1)$ extracted bits and therefore there are $O(n^4)$ possible distances among these bits. Here we do this: we find an available distance d_1 and do $b_{i/2} = b_i \vee (b_{i+1} \to d_1)$. We have 2 integers $\vee$-ed into 1 integer. Thus we have now $n/2$ integers remain. Instead of continuing $\vee$-ing integers together we now extract half of the bits corresponding to indices $m'(i)$, $i = 0, 1, ..., (n-1)/2 - 1$ to one integer and another half of the bits corresponding indices $m'(i)$, $i = (n-1)/2, (n-1)/2 + 1, ..., n-1$ to another integer. Before extracting the number of possible distances in the integer is $2(n-1)^2$. After extracting them into 2 integers each integer has the number of possible distances $2((n-1)/2)^2 = (n-1)^2/2$ (each $\vee$-ed in integer has only $(n-1)/2$ bits now). The situation is basically the same as the analysis we had for the $O(n^3)$ possible distances. This is equivalent to say that we have put 2 integers into 2 integers and the number of possible distances among bits decreased from $(n-1)^2$ to $(n-1)^2/2$ for each integer. However, for the further $\vee$-ing together integers we have now to pick a d_2 for $(n-1)/2$

integers and pick another d'_2 for the other $(n-1)/2$ integers and thus the number of possible distances is $(n-1)^2/2+(n-1)^2/2=(n-1)^2$. We can repeat this $\log n$ times, each time $\vee$-ing 2 integers into 1 integer and then extract out 2 integers from this integer. Every time we extract 2 integers the number of $m'(i)$'s the bits corresponds to in an integer is divided by 2. After $\log n$ times the number of $m'(i)$ the bits in an integer correspond to is reduced to 1, i.e. the $m'(i)$-th bits of all integers are now in one integer and the $m'(j)$-th bits of all integers for $j \neq i$ are now in another integer. We repeated $\log n$ times and each time we have to spend $O(n^2)$ time searching for d_i's. Thus the overall time of our algorithm is $O(n^2 \log n)$.

Theorem 2: For integers of $\Omega(n^2)$ bits, a perfect hash function can be constructed in $O(n^2 \log n)$ time for hashing n integers to $O(n)$ bits integers (i.e. pack the $n-1$ bits with indices in $\{m'(0), m'(1), ..., m'(n-1)\}$ to $n-1$ bits in $O(n^2 \log n)$ time). Thereafter the packing and hashing of a batch of n integers to $O(n)$ bits integers take $O(n \log n)$ time. $\square$

Note that although we packed the extracted bits to $n-1$ bits and therefore it seems that $\log m = n-1$ in Theorem 2. However our algorithm assumed that integers having $\Omega(n^2)$ bits. Thus for integers of $O(n^2)$ bits Raman's hash function construction algorithm would require $O(n^2 \log m) = O(n^4)$ time.

Note also that although Theorem 2 reduced the computing time for finding $d_1, d_2, ...$ to $O(n^2 \log n)$ it has the disadvantage of packing and hashing a batch of n integers in $O(n \log n)$ time instead of the $O(n)$ time we have achieved in the last section. In the next section we will show how to reduce the time for computing $d_1, d_2, ...,$ to $O(n^2 \log^2 n)$ time while keeping the packing and hashing time to $O(n)$.

5 Achieving $O(n)$ Packing Time with $O(n^2 \log^2 n)$ Time for Construction

Here we first compute d_j and do $b_{i/2} = b_i \vee (b_{i+1} \rightarrow d_j)$, $i = 0, 2, 4, ...$, for $j = 1, 2, ..., \log\log n$. We have thus $\vee$-ed $\log n$ integers into one integer and we have only $n/\log n$ integers remain. Now the number of possible distances in each integer becomes $(\log n)(n-1)^2$. We now take over the algorithm in last section. That is we will $\vee$ two integers into one integer and then extract two integers from one integer. We need to do this $O(\log n)$ times. In doing so the total number of distances is kept at $(\log n)(n-1)^2$ and each time we do this we spent $O(n/\log n)$ time as we have only $n/\log n$ integer. This makes the overall time for packing become $O(n)$. The time for computing distances $d_1, d_2, ...$ becomes $O(n^2 \log^2 n)$ as we repeated $O(\log n)$ times and each time expending $O(n^2 \log n)$ time.

Theorem 3: For integers of $\Omega(n^2 \log n)$ bits, a perfect hash function can be constructed in $O(n^2 \log^2 n)$ time for hashing n integers to $O(n)$ bits integers (i.e. pack the $n-1$ bits with indices in $\{m'(0), m'(1), ..., m'(n-1)\}$ to $n-1$ bits in $O(n^2 \log^2 n)$ time). Thereafter the packing and hashing of a batch of n integers to $O(n)$ bits integers take $O(n)$ time. $\square$

Here we require that integers have $\Omega(n^2 \log n)$ bits. For integers of $O(n^2 \log n)$ bits Raman's hash function construction takes $O(n^2 \log m) = O(n^4 \log n)$ time.

Main Theorem: A perfect hash function for n integers can be constructed in $O(n^4 \log n)$ time. Thereafter hashing a batch of n integers takes $O(n)$ time. $\square$

Because the main operations in our algorithms are ∨, ∧, and shift operations and because the algorithm of Dietzfelbinger et al. [5] are Raman's derandomization [14] is a word hash algorithm and thus our algorithms achieved are word hashing algorithms. Thus we have:

Corollary to the Main Theorem: A perfect word hash function for n integers can be constructed in $O(n^4 \log n)$ time. Thereafter hashing a batch of n integers takes $O(n)$ time. □

Such a word hash function is applied in the construction of our ordered partition algorithm [11].

Acknowledgment

I would like to thank M. Dietzfelbinger and T. Hagerup for bringing Ruzic's paper [16] and Thorup's paper [17] to my attention. I would also like to thank the reviewers for their careful review work.

References

1. A. Andersson, T. Hagerup, S. Nilsson, R. Raman. Sorting in linear time? *Proc. 1995 Symposium on Theory of Computing STOC'1995*, 427-436(1995). Also in *Journal of Computer and System Science* **57**, 74-93(1998).
2. Fabiano C. Botelho, Rasmus Pagh, Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Inf. Syst.* 38(1): 108-131 (2013).
3. D. Belazzougui, G. S. Brodal, J. S. Nielsen. Expected linear time sorting for word size $\Omega(\log^2 n \log\log n)$. In SWAT'2014. LNCS 8503, 26-37(2014).
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, 3rd Edition, The MIT Press, 2009.
5. M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms* **25**, 19-51(1997).
6. M. L. Fredman, J. Komlós, E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, Vol. 31, No. 3, 538-544(1984).
7. M. L. Fredman, D. E. Willard.Surpassing The Information Theoretic Barrier with Fusion Trees *Invited paper in Journal of Computer and System Sciences* 47, 424-433(1993).
8. T. Hagerup, Peter Bro Miltersen, Rasmus Pagh. Deterministic dictionaries. *J. Algorithms* 41(1): 69-85 (2001)
9. Y. Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *Journal of Algorithms*, 50, 96-105(2004).
10. Y. Han. Improved fast integer sorting in linear space. *Information and Computation*, Vol. 170, No. 1, 81-94(Oct. 2001).
11. Y. Han. A linear time algorithm for ordered partition. *Proceedings of 2015 International Workshop on Frontiers in Algorithmics (FAW'2015)*, LNCS 9130, 89-103(2015).
12. Y. Han, M. Thorup. Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space, *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science(FOCS'02)*, 135–144(2002).
13. Rasmus Pagh. Hash and displace. Efficient evaluation of minimal perfect hash functions. *WADS* 1999: 49-54.
14. R. Raman. Priority queues: small, monotone and trans-dichotomous. *Proc. 1996 European Symp. on Algorithms, Lecture Notes in Computer Science 1136*, 121-137(1996).
15. M. Ruzic. Uniform deterministic dictionaries. *ACM Transactions on Algorithms* 4(1) (2008).
16. M. Ruzic. Constructing efficient dictionaries in close to sorting time. *ICALP 2008*, Part 1, LNCS 5125, pp. 84-95.
17. M. Pătraşcu, M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. *Proc. 2014 IEEE Symp. on Foundations of Computer Science*, 166-175(2014).