BEATING THE POPCOUNT

Eyas El-Qawasmeh
Dept. of Computer Science and Information Systems
Jordan University of Science and Technology
P.O. Box 3030
Irbid 22110, Jordan
eyas@just.edu.jo
http://www.cis.just.edu.jo/People/Faculty/eyas/index.htm

Abstract

Popcount is a built-in function that was implemented using some extra hardware in many computers. Its purpose is to count the number of ones in a given computer word. This built-in function was able to beat other software implementations such as serial shifting. In this paper, we propose a software algorithm of this function that achieves the same purpose without the need for any extra hardware. The suggested algorithm is faster than the hardware implementation of popcount. In addition, the performance of the suggested algorithm is investigated and evaluated.

Key words. Bit-Counting, Bit-Parallelism, Counters, Document Retrieval, Frequency Division, Performance, Popcount.

1. INTRODUCTION

Given an array of binary vectors, it is often necessary to determine how many "1" bits the array contains. This problem, which is called the population count problem or bit-counting problem, is abbreviated by popcount. It is used in many applications such as information retrieval, file processing, and coding theory (Berkovich et al., 2000). For example, consider an information system such as a database where the relevant information is presented in terms of "1" and the non-relevant information in terms of "0". In this case, the bit-counting can be used to respond to the user queries. Popcount was used for SQL run optimization, when joining bitmaps to determine the best evaluation order, and to implement function count-trailing-zeros( ), which are used heavily in the binary **GCD** (greatest common divisor). Other examples include a comparison between a query and a document signature during the search for a query answer, and finding the difference between two documents. All the previous examples can benefit from reducing the retrieved time of queries.

The popcount operation was implemented using extra hardware with low cost. It was introduced in the Control Data's CDC 6000 series that had a "count ones" instruction that worked on 60-bit word. After that, Gray Research, Inc, uses a 64-bit hardware implementation to perform the popcount upon the request of one customer, and then added by other companies such as Digital, and HP.

On the other hand, popcount can be implemented using software without the need for any special hardware. Currently, there are several software implementations for the popcount function, although the word popcount is not used. These implementations are: a) Serial shifting, b) Arithmetic logic counting (AL), c) Emulated popcount, d) Lookup table e)-Hamming distance bit vertical Counter (HC), and f) Frequency division (Berkovich et al., 2000)(Berkovich et al., 1998)(El-Qawasmeh, Hemidi, 2000)(Gutman, 2000)(Reingold et al., 1977).

The main objective of this paper is to present a new technique for popcount that is based on the frequency division principle. In addition, the performance analysis of the suggested technique will be investigated and compared with the hardware implementation of popcount. Experimental results showed that the proposed scheme is the best among all known techniques, whether they are implemented in software or hardware.

This paper is organized as follows: Section 2 describes current related work. Section 3 describes the hardware implementation. Section 4 is the suggested count technique.

Section 5 is remedy improvements. Section 6 is the performance experimental results, followed by conclusions in section 7.

2. CURRENT RELATED WORK

The popcount operation takes an array of binary vectors, A[I],(I=1,…,n), and returns the number of "1"s the array contains. Despite the apparent simplicity, this problem is associated with interesting algorithmic issues. The popcount operation has been considered so intuitive that it was frequently performed without any research required. The methods used were simple in concept with varying degrees of efficiency. The details of these schemes are out of the scope of this paper, but they can be found in (El-Qawasmeh, 2001). However, we will list these schemes very briefly with an explanation for two of them that are necessary for clarifying our technique in this paper (The Hamming distance bit vertical counter and the frequency division, since our algorithm is based on them).

The first, and simplest scheme for popcount is serial shifting, which is implemented as follows:

```
Counter = 0;
While the Number ≠ 0 do
    If the lowest bit of Number is 1 then
        Increase Counter by one.
        Shift the Number to the right one bit.
    End If
End while
```

The second implementation of popcount is called Arithmetic Logic counting (AL). This method depends on doing the mask operation (AND) for a number with itself after subtracting one from it. The same logic operation is repeated as long as the number does not equal to zero. Further details can be found in (El-Qawasmeh, Hemidi, 2000) (El-Qawasmeh, 2001).

The third implementation is called emulated popcount. The emulated popcount algorithm successively groups the bits into sub-groups of 2, 4, 8, 16, and 32, while maintaining a count of "1"s in each group.

The fourth implementation uses Lookup tables. This algorithm constructs static table(s) only once. In a 32-bit machine we construct a single table of size equal to 2^{32}. Once the table has been defined, each element will be filled with the number of "1"s that exists in the index of the element. Therefore, to determine the number of “1”s for any given 32-bit computer word, we use the given 32-bit computer word as an index to the table. Therefore, a single read statement is enough.

However, a table of size equal to 2^{32} elements might be a restriction. Therefore, it is possible to reduce the size of the table to 2^{16}. Although the reduction is an advantage, it is accompanied with a disadvantage that the 32-bit computer word is divided into two 16-bit groups where each 16-bit will be an index to the table. Thus, two read statements, and an extra splitting operation are needed.

The fifth implementation is called Hamming distance bit vertical Counter (HC). This algorithm finds the total number of “1”s for a collection of vectors rather than one vector. However, for a single vector, we can make the number of the vectors in the collection to be equal to one. It uses a half adder calculation form to determine the total through the sum and carry values principle. Each vector use the following two formulas

sum = vector $\oplus$ carry   where $\oplus$ is an **Exclusive OR** (XOR)

and

carry = vector * carry   where * is an **AND**

where sum, vector, and carry are binary vectors consisting of n-bit. As an example, consider a set of 4 vectors each one consists of 4-bits (see Figure 1). At the start of the calculation, the Exclusive **OR**, will be applied between the carry, which is zero, and the first vector so that the result will be stored in the vertical counter (An array of size equal to (or less than) $\log_2$ (No. of vectors in the collection)). After that, the second vector will be added to the generated carry resulted from the previous sum step using the same half adder procedure, and so on. Once all the vectors are exhausted, the answer will be in a vertical format. Now, for all the elements in the collection, a single serial shift will be used to pull the answer into a horizontal form.

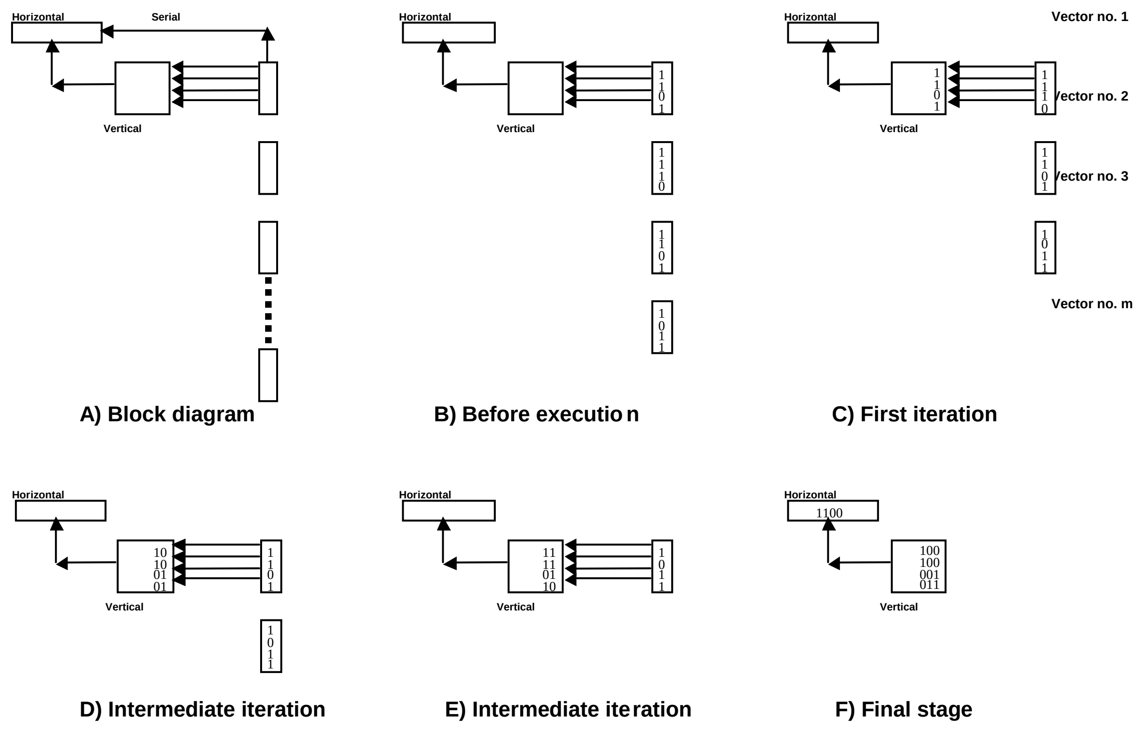


Figure 1: Popcount using Hamming distance bit vertical counter

The reader should be aware that the example given in figure 1 is for illustration purposes and might not show clear time saving. However, if the number of vectors in a collection is large, then more saving will be gained. Figure 2 shows the exact details of the Hamming distance bit vertical Counter (HC) (El-Qawasmeh, Hemidi, 2000).

```
For (each bit in the vector) do                    // vector is the number in binary
        Set Vertical_Counter[bit] = 0
End for
Set Total_Number_of_Ones  = 0
While (there are more vectors) do
        Set Carry = vector
        Set Index  = 0
        While ((Carry ≠ 0) and (Index < No_of_bits in the vector)) do
                Set Temp = Vertical_Counter[Index]
                Set Vertical_Counter[Index] = Temp ⊕ Carry where ⊕  stands for Exclusive OR
                Set Carry = Temp * Carry where * is AND operation
                Increment Index by 1
        End While
End While
For (Index = No_of_bits in the vector down to 1) do
        Shift Total_Number_of_Ones to the left one bit
        Set Temp = Vertical_Counter[Index]
        If (Temp ≠ 0) then
                For (each bit in the vector) do
                        If (the most significant bit of Temp is “one”) then
                                Increment Total_Number_of_Ones by 1
                        End If
                        Shift Temp one bit to the right
                End For
        End If
End For
```

Figure 2: The algorithm for counting the number of “1”s using the HC

The sixth approach is the Frequency Division (FD) technique. This algorithm is designed to find the total number of “1”s for a set of vectors. However, it can work for a single vector. This method uses the Hamming distance bit vertical Counter (HC), however, it is improved by incorporating Arithmetic Logic (AL) algorithm. The new algorithm is called the Frequency Division algorithm (Berkovich et al., 2000). The FD technique consists of two stages. In the first stage, the incoming binary vectors are processed in the same way as in the vertical counter of some fixed number of columns. In the second stage, the bits passing through the Low-bit Sieve (The rightmost bits of HC) are counted by the arithmetic logic (AL) scheme mentioned previously.

The HC part of the Frequency Division technique is performed with a small number of relatively fixed columns in Low-bit Sieve. The AL part of the algorithm is performed by dropping bits for every 2^K incoming “1” bits. A complete listing of this algorithm can be found in (Berkovich et al., 2000). The algorithm that implements the FD for k=2 is given below:

```
Set $V_0$ = 0; $V_1$ = 0; Set Total_Number_of_Ones = 0
**While** (there are more vectors) **do**
        Set $carry_1 = V_0$ & vector          // & represents AND operator
        Set $V_0 = V_0$ ^ vector              // ^ represents Exclusive OR (XOR)
        Set $carry_2 = V_1$ & $carry_1$
        Set $V_1 = V_1$ ^ $carry_1$
        **While** ($carry_2 \neq 0$) **do**
                Set $carry_2 = carry_2$ & ( $carry_2$ -1)
                Increase Total_Number_of_Ones by four
        **End While**
**End While**
**While** ($V_1 \neq 0$) **do**
        Set $V_1 = V_1$ & ($V_1$ -1)
        Increase Total_Number_of_Ones by two
**End While**
**While** ($V_0 \neq 0$) **do**
        Set $V_0 = V_0$ & ($V_0$ - 1)
        Increase Total_Number_of_Ones by one
**End While**
```

Figure 3: Frequency Division algorithm for K= 2 (FD2)

3. HARDWARE IMPLEMENTATION

The built-in popcount function allows you to use directly the supported hardware designed for it and the machine instructions. It is available in a limited number of programming languages such as C/C++. The syntax for calling this function varies from one language to another. For example in C/C++, the call to this function is achieved by writing "_popcnt(unsigned integer)". To do this in C/C++, you must include the <machine/builtins.h> header file in your source program to access this function. This intrinsic function is processed completely by the compiler. Because "_popcnt" is an intrinsic function, which is executed using the designated hardware circuitry, no externally visible library function is available for it. The compiler generates inline code, which is one or two instructions in many cases, to produce the result. It is called a function because it is invoked with the syntax of function calls (El-Qawasmeh, 2001).

A vector version of the popcount intrinsic function exists on some machines such as UNICOS and UNICOS/mk systems. On UNICOS/mk systems, the vector version of this intrinsic is used when **-O vector3** or **-O 3** is specified on the compiler command line. If a vector version of an intrinsic function exists and the intrinsic is called within a vectorized loop, the compiler uses the vector version of the intrinsic.

The difference between all these algorithms (Software and hardware implementations) is in the execution time of the popcount. In addition, some other factors might affect the performance, for example, the nature of computer words. By this we mean if the binary representation of the numbers has the majority of its bits “zeros” or “ones”.

To measure the performance of all the previous schemes we simulate a system by creating several tens of thousands of 64-bit binary vectors using Unix machine. The execution time for these vectors was measured after they are loaded into the memory using all the previous software implementations and also the hardware implementation of the popcount. For reliable timing, the execution was repeated several hundreds of times, and the average execution time was considered. The creation of these vectors was done with a certain probability of “1”s that range from zero to one. Each bit in any given vector was generated randomly and independently from all other bits. The random number generator that was used can be found in (Press et al., 1995). To understand the generation process, consider any probability such as 0.25, then the probability that a given bit in a 64-bit vector is “1” will be 0.25. This means that the majority of the 64-bit binary vectors will have 16 “ones” on average. This is helpful in order to relate the nature of the binary vectors with the performance. Using the same sets of vectors with different probabilities of “1”s, the computation time for each algorithm is measured. Figure 4 shows a comparison between all these techniques except serial shifting (Serial shifting was omitted since it is the worst among all of them).

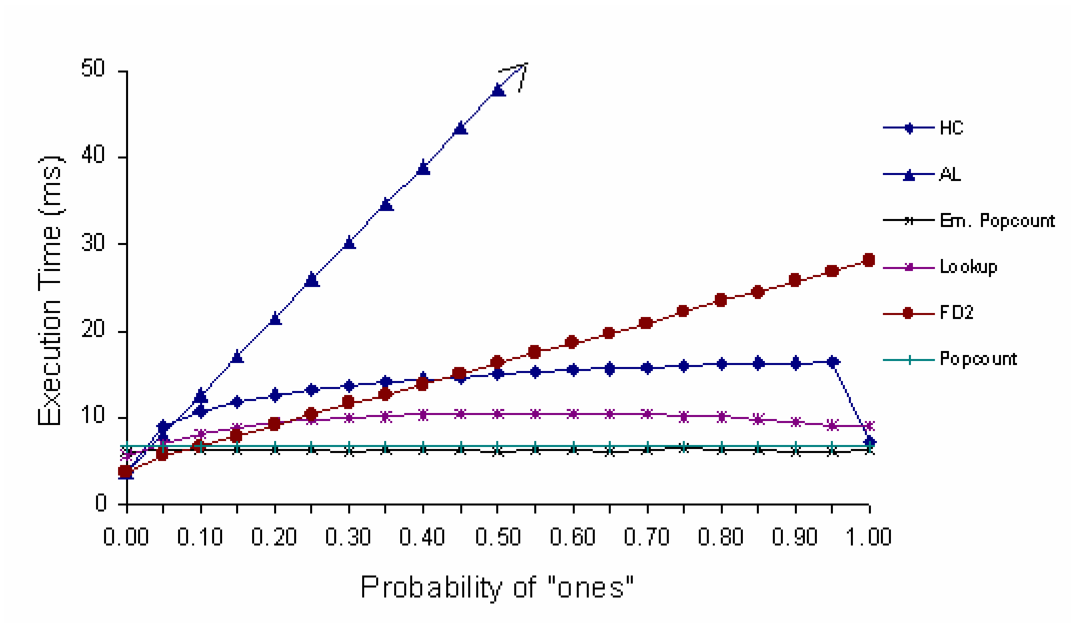


Figure 4: Time comparison between different counters vs. probability of “1”s using a 64-bit machine

To understand Figure 4 let us select a probability of 0.5. This means that most of the items in a database will have about 50% of the computer word as “1”. In this case, the AL execution time is more than the emulated popcount by a factor of 6 approximately. For HC, it will need about 2.5 times more than the emulated popcount. In Figure 4 and all the following figures, the time was computed using the built-in clock() function at the beginning and at the end of each algorithm. The results of this experiment and all following experiments are normalized to the execution time of the built-in hardware implementation of popcount. Also, the timing is averaged over hundreds of runs for this figure and all the following figures. Figure 4 shows that the emulated popcount is the best one, and its execution time is very close to the built-in function popcount that uses a special hardware. However, for very sparse/dense systems of “1”s, FD procedure will be better than the emulated popcount.

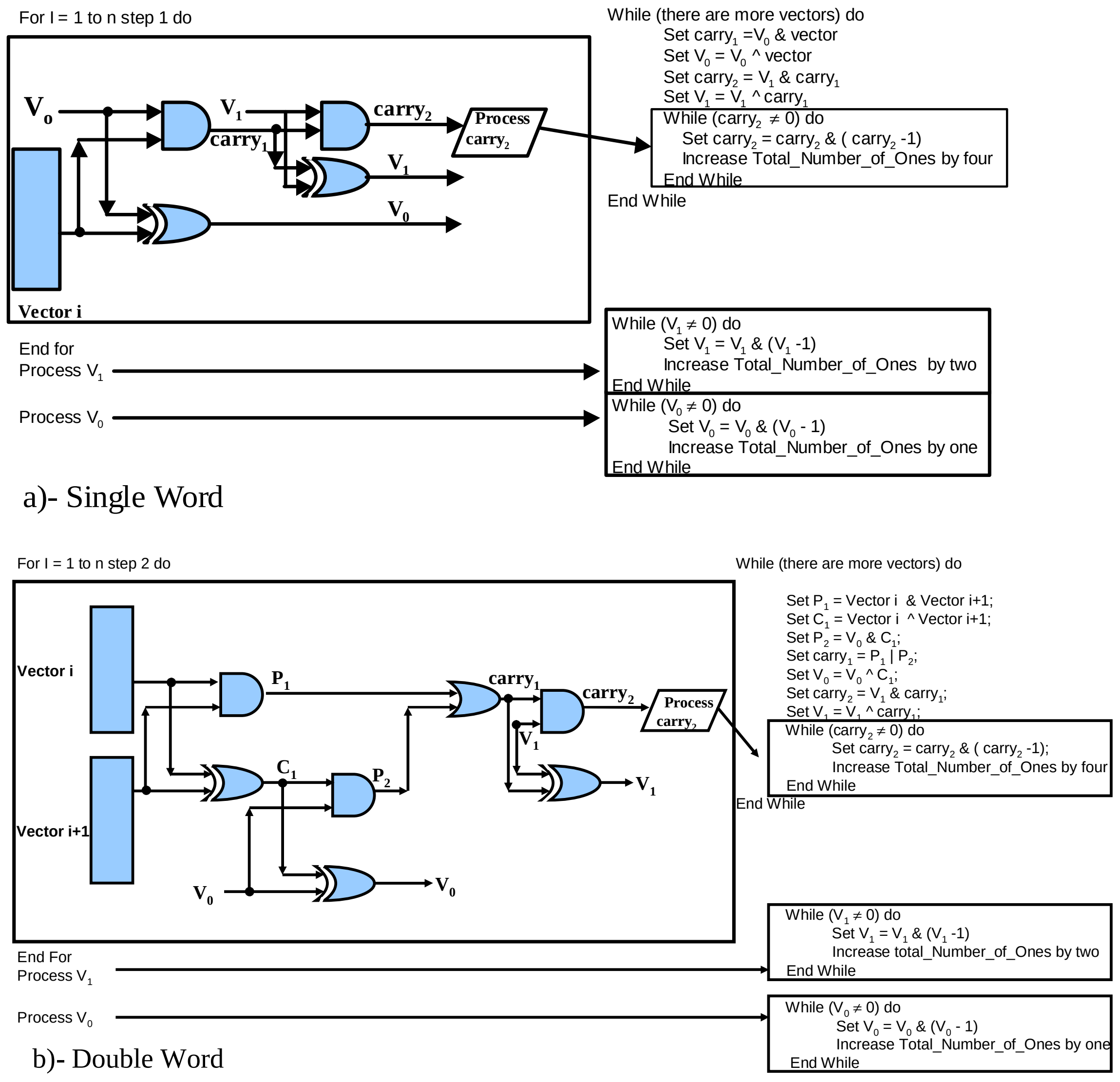
As can be seen from Figure 4, we have FD2 although using the Frequency Division principle we can have FD3, FD4, and so on. The difference between them is in the timing that a count operation occurs. For example, in FD2, we do the count operation when two bits filled with “1”s are encountered while for FD4 we do the count operation when four bits filled with “1”s are encountered. The same idea is applicable to FD5, FD6, and FD7.

In FD family (FD2, FD3, FD4,…..), the time to do the counting operation has a relation with the number of vectors in the system. For example, suppose that you are using FD7 when the number of vectors in your system is 128 vectors. Then you need only $\log_2 128 = 7$ bits for the vertical counter. Now if the size of your system is increased by a factor of

1000. i.e., it becomes 128000 vectors, then you still need only 7-bits, but you need to do the count operation every k^2= 128 vectors approximately.

4. SUGGESTED COUNT TECHNIQUE

The suggested technique is based on the FD algorithm. The FD scheme is always handling one vector (computer word) at a time into the vertical counter. For this purpose, we will call the previous FD scheme a single word FD scheme. Our suggested technique is to handle two words at the same time instead of one. Handling two words will be called double word scheme. Figure 5 shows the difference between single and double word for FD2.



```
For I = 1 to n step 1 do
End for
Process $V_1$
Process $V_0$
```

```
While (there are more vectors) do
        Set $carry_1 = V_0$ & vector
        Set $V_0 = V_0$ ^ vector
        Set $carry_2 = V_1$ & $carry_1$
        Set $V_1 = V_1$ ^ $carry_1$
        While ($carry_2 \neq 0$) do
           Set $carry_2 = carry_2$ & ( $carry_2$ -1)
           Increase Total_Number_of_Ones by four
        End While
End While
```

```
While ($V_1 \neq 0$) do
        Set $V_1 = V_1$ & ($V_1$ -1)
        Increase Total_Number_of_Ones  by two
End While
While ($V_0 \neq 0$) do
        Set $V_0 = V_0$ & ($V_0$ - 1)
        Increase Total_Number_of_Ones by one
End While
```

```
For I = 1 to n step 2 do
End For
Process $V_1$
Process $V_0$
```

```
While (there are more vectors) do
        Set $P_1$ = Vector i  & Vector i+1;
        Set $C_1$ = Vector i  ^ Vector i+1;
        Set $P_2 = V_0$ & $C_1$;
        Set $carry_1 = P_1$ | $P_2$;
        Set $V_0 = V_0$ ^ $C_1$;
        Set $carry_2 = V_1$ & $carry_1$;
        Set $V_1 = V_1$ ^ $carry_1$;
        While ($carry_2 \neq 0$) do
                Set $carry_2 = carry_2$ & ( $carry_2$ -1);
                Increase Total_Number_of_Ones by four
        End While
End While
```

```
While ($V_1 \neq 0$) do
        Set $V_1 = V_1$ & ($V_1$ -1)
        Increase total_Number_of_Ones by two
End While
While ($V_0 \neq 0$) do
        Set $V_0 = V_0$ & ($V_0$ - 1)
        Increase Total_Number_of_Ones by one
End While
```

Figure 5: FD2 Block diagram of single and double word

Figure 5 shows our suggested technique using double word. In this technique, no logic gates exists, since all of them are implemented using the code on the right hand side of Figure 5. The logic gates serves only to count the number of operations required in our suggested technique.

Given any **n** numbers, if we apply the FD2 using a single word then the main block of Figure 5.a will be executed **n** times since we are handling one vector at a time. However, in Figure 5.b, we will execute the main block only **n/2** since we are handling two words at the same time.

Thus, there will be a saving of one operation. In Figure 5.a, we will need $4n$ operations (each gate represents one operation), while in Figure 5.b we need only $7\frac{n}{2}=\frac{7n}{2}$ operations. Although, it is not big saving, but if we use double word with higher FDi's, then larger saving will occur.

From Figure 3 and Figure 5, we see that we need to process carry$_2$ box (presented by ▱ Figure 5). However we can add a test (if statement) that avoids the while loop if the contents of carry$_2$ is zero. This has a small saving on FD2, but if we proceed to other FDi's, then larger saving will be gained.

We applied the double word suggestion to FD2, FD3, FD4,….,FD7, meanwhile we use to check whether the first carry is zero or not in order to avoid the while loop. In each case of FDi's, we have verified that using double word would take less time than single word. From these FDi's results, we selected FD5, which is depicted in Figure 6

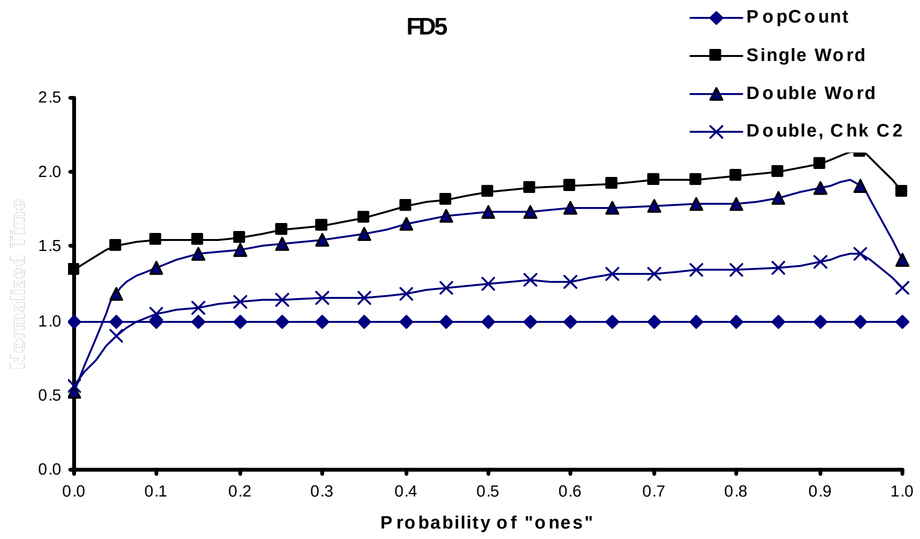


Figure 6: FD5 using double word and check for the first carry

Figure 6 shows the execution time of FD5 and the built-in popcount function. The execution time of the built-in popcount function was constant regardless of the number of “1”s in our vector. In Figure 6, the FD5 was applied with a single word, double word, and double word that checks if the $carry_2$ is zero. If $carry_2$ is zero, then the rest of the while loop will be skipped. Figure 6 verifies that using double word with check for carry provides the best execution time.

A further improvement is to use 4 words at the same time instead of 2 words. In this case, the main block in Figure 5 part a will be repeated 3 times instead of 1. However, we will process 4 words instead of single word. Experiment for this was done on FD2, FD3,…FD6, FD7. Results of all of them verified that moving to 4 words is much better than double word.

Following is the algorithm for FD5 using Quadruple word (4 words). This algorithm can also be applied to other FDi’s with slight modifications.

```
$V_0$  = 0; $V_1$ = 0;$V_2$ = 0;$V_3$=0; $V_4$=0; $V_5$=0;Counter = 0;
/* The meaning of the following symbols is as follows        */
/*   &  : The logic AND operation                             */
/*   ^  : The logic Bitwise Exclusive OR                      */
/*   |  : The logic OR operation                              */
/* Propi: Denote a propagate with associated number           */
/* Ci   : Indicate a carry associated with a number           */
For (J=0;J<No_of_Elements - 3;J=J+4) do
        $Prop_1$ = Value[J] & Value[J+1];
        $C_1$ = Value[J] ^ Value[J+1];
        $Prop_2$ = $V_0$ & $C_1$;
        $carry_1$ = $Prop_1$ | $Prop_2$;
        $V_0$ = $V_0$ ^ $C_1$;              // The HC part
        /**** First two words ****/
        $Prop_3$ = Value[J+2]&Value[J+3];
        $C_2$ = Value[J+2]^Value[J+3];
        $Prop_4$ = $V_0$ & $C_2$;
        $carry_2$ = $Prop_3$ | $Prop_4$;
        $V_0$ = $V_0$ ^ $C_2$;
        //**** Second two words ****/
        Prop5 = $carry_1$ & $carry_2$;
        Merged_Cs = $carry_1$  ^ $carry_2$;
        $Prop_6$= $V_1$ & Merged_Cs;///
        Merged_Carrys = Prop5 | Prop6;
        $V_1$= $V_1$ ^ Merged_Cs;
        //**** Result of first level ****/
        $carry_3$ = $V_2$ & Merged_Carrys;
        $V_2$ = $V_2$ ^ Merged_Carrys;
        $carry_4$ = $V_3$ & $carry_3$;
        $V_3$ = $V_3$ ^ $carry_3$;
        $carry_5$ = $V_4$ & $carry_4$;
        $V_4$ = $V_4$ ^ $carry_4$;
        While($carry_5 \neq 0$)  do
                /* The AL part counting every 32th bit */
                $carry_5 = carry_5 \& (carry_5 - 1)$;
                Counter= Counter + 32; // Increment counter by 32
        End While

End For
/* Appending the contents of LS*/
While ($V_4 \neq 0$) do
{       $V_4 = V_4 \& (V_4 - 1)$; Counter = Counter + 16;}
While ($V_3 \neq 0$) do
{       $V_3 = V_3 \& (V_3 - 1)$; Counter = Counter + 8;}
While ($V_2 \neq 0$) do
{       $V_2 = V_2 \& (V_2 - 1)$; Counter = Counter + 4;}
While ($V_1 \neq 0$) do
{       $V_1 = V_1 \& (V_1 - 1)$; Counter = Counter + 2;}
While ($V_0 \neq 0$) do
{       $V_0 = V_0 \& (V_0 - 1)$; Counter = Counter + 1;}
```

Figure 7: FD5 with Quadruple words code

We continue in this process by handling every 8 words, then handling every 16 words instead of 4 words. This was done by changing the step increment in the for loop and modifying the algorithm listed in Figure 7. Results showed that using 16 words is better than using 2, 4, or 8 words. This has been verified with all FDi's ranging from FD2 to FD7. However, if we fix the number of treated words, say 16 words, then the execution time of FD7 will be better than the execution time of FD6 always. FD6 is better than FD5 and so on. In fact, the execution time for FD7, and F6 was always below the hardware implementation of popcount when we used 16 words. For FD5, it was most of the time. As an example, we have selected both FD5 and FD6 that are depicted in Figure 8.

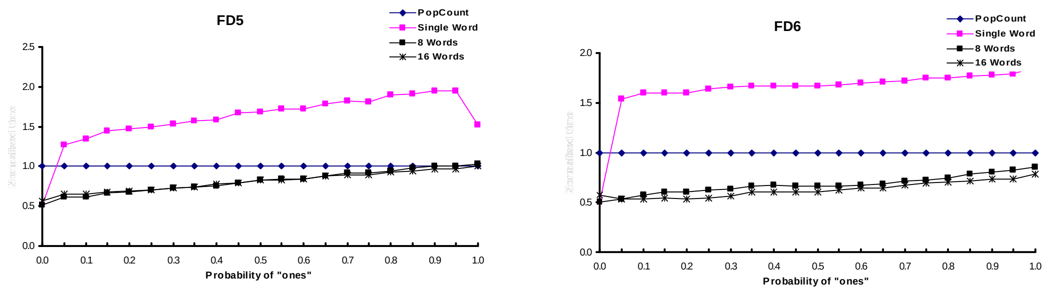


Figure 8: Comparison between FD5 and FD6 shows that higher FDi is better

Figure 8 shows clearly that using 16 words in FD6 will achieve the minimum execution time. This time is below the popcount time no matter what probability of "ones" we have. The new execution time will be 20% - 50% less than the popcount implemented in hardware. In case, the user decided to use FD7, then more saving will be achieved.

5. REMEDY IMPROVEMENTS

As we saw in the last section we are able to get an execution time that is less than the hardware implementation of popcount function. However, if we merge the hardware implementation of popcount with our technique then we will improve the performance. Thus we will use our technique over 16 words; meanwhile, we will use the hardware implementation of the popcount inside the loop of our technique. Thus, for every 16 words that we handle, we will use the hardware implementation to find the number of ones for $carry_5$ ($carry_5$ is the normal carry at the fifth level). Note that within our technique, we will use the hardware implementation of popcount 16 times since we are dealing with 16 different words.

According to the above mentioned, we will replace the following while loop (listed in Figure 7)

```
While ($carry_5 \neq 0$) do                // $carry_5$ is the normal carry at the fifth level
        /* The AL part counting every 32th bit */
        $carry_5$ = $carry_5$ & ( $carry_5$ -1);
        Counter = Counter + 32;// Increment counter by 32
End While
```

by a popcount statement. Thus, we will have:

```
        Counter = Counter + popcount($carry_5$) * 8
        // Since the AL part counts every 32th bit, then consider $\log_2 32 = 5$ bits. This 5 //
        // bits will be presented by $(10000)_2$ which is equal to 16 //
```

Using the previous explanation we will be able to remove the while loop. The same technique that we use for finding the popcount of $carry_5$ will be applied to V_i's. Thus, for 8 words and FD5 we replace all the following statement:

```
While ($V_4 \neq 0$) do
{       $V_4 = V_4$ & ($V_4$ -1); Counter = Counter + 16;}
While ($V_3 \neq 0$) do
{       $V_3 = V_3$ & ($V_3$ -1); Counter = Counter + 8;}
While ($V_2 \neq 0$) do
{       $V_2 = V_2$ & ($V_2$ -1); Counter = Counter + 4;}
While ($V_1 \neq 0$) do
{       $V_1 = V_1$ & ($V_1$ -1); Counter = Counter + 2;}
While ($V_0 \neq 0$) do
{       $V_0 = V_0$ & ($V_0$ -1); Counter = Counter + 1;}
```

by the following statement:

Counter = Counter + popcount(V_4)* $\log_2 16$ + popcount(V_3) $\log_2 8$ + popcount (V_2) $\log_2 4$ + popcount(V_1) * $\log_2 2$ + popcount (V_0) * $\log_2 1$.

After finding the $\log_2$ values and using Horner's rule, we will be able to reduce the last equation to:

$$\text{Counter} = \text{Counter} + 2\,(\text{popcount}(V_0) + 2\,(\text{popcount}(V_1)+2\,(\text{popcount}(V_2)+2\,(\text{popcount}(V_3)+2\,(\text{popcount}(V_4))))$$

The last equation provides the minimum number of operations. In this equation the hardware implementation of popcount was combined with our method. However, for a set of 16 words, we call the hardware implementation of popcount only once, instead of 16 calls.

6. PERFORMANCE RESULTS

Experimental results of merging the hardware implementation of popcount with our technique give us an execution time that is almost half the execution time with the hardware implementation of popcount. Figure 9 presents the results of the execution time.

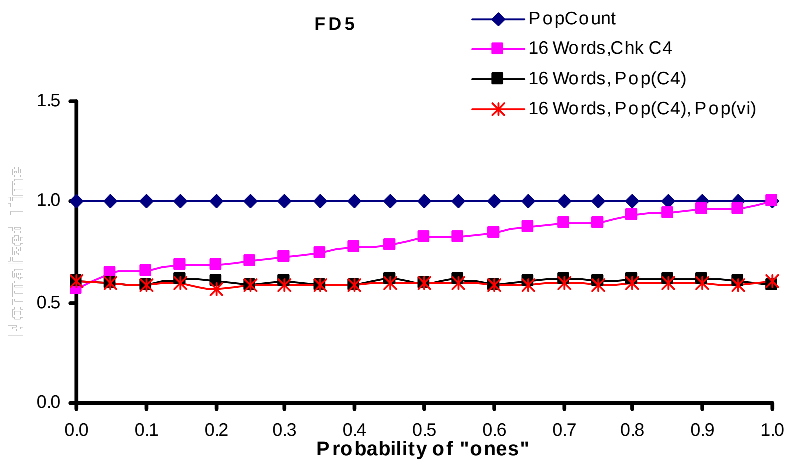


Figure 9: The execution time for FD5 using 16 words technique and the hardware implementation of popcount

Another experiment was conducted to see when our approach would not be able to beat the popcount if we change the number of words we are handling at the same time. We run our algorithm to handle 2, 4, 8, and 16 words with FD2, FD3,…, FD7, meanwhile, we did not use the hardware implementation of popcount. Results showed that the behavior of FD_i family is close. As an example, we have selected FD5

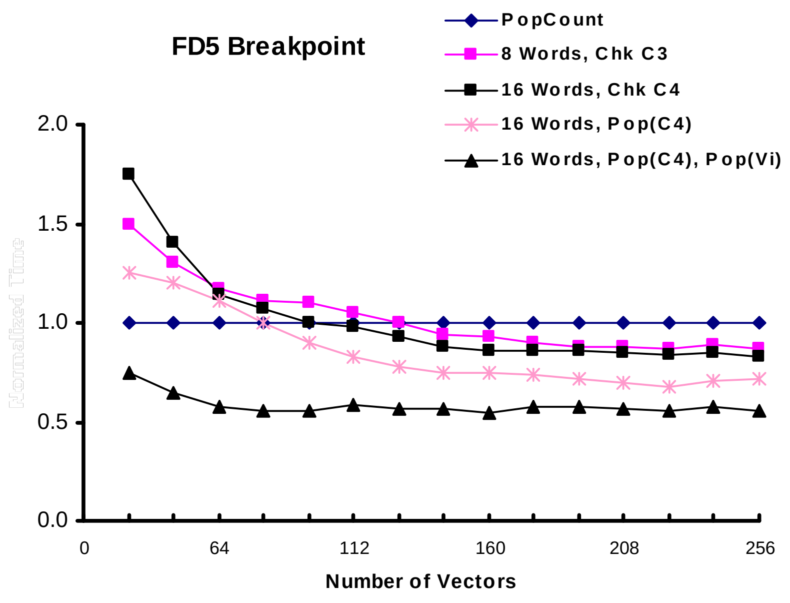


Figure 10: Break points for FD5 with the built-in popcount

As can be seen from Figure 10, our algorithm will be able to beat the popcount when the number of the values we are interested in finding their popcount exceeds 120 values approximately. In our suggested technique we never use the hardware implementation of popcount. However, merging our technique with hardware implementation of popcount for 16 words gives us less execution time than using hardware implementation of the popcount alone or the software implementation.

The last experiment that was conducted is to determine the speedup gained from using our technique combined with the hardware implementation of popcount against single word technique. We checked this for 2, 4, 8, and 16 words merged with the popcount technique for the carry. Results showed that our technique would be faster than the built-in popcount function in most of the cases. The speedup was maximized when we used 16 words. Results of these experiments are depicted in Figure 11.

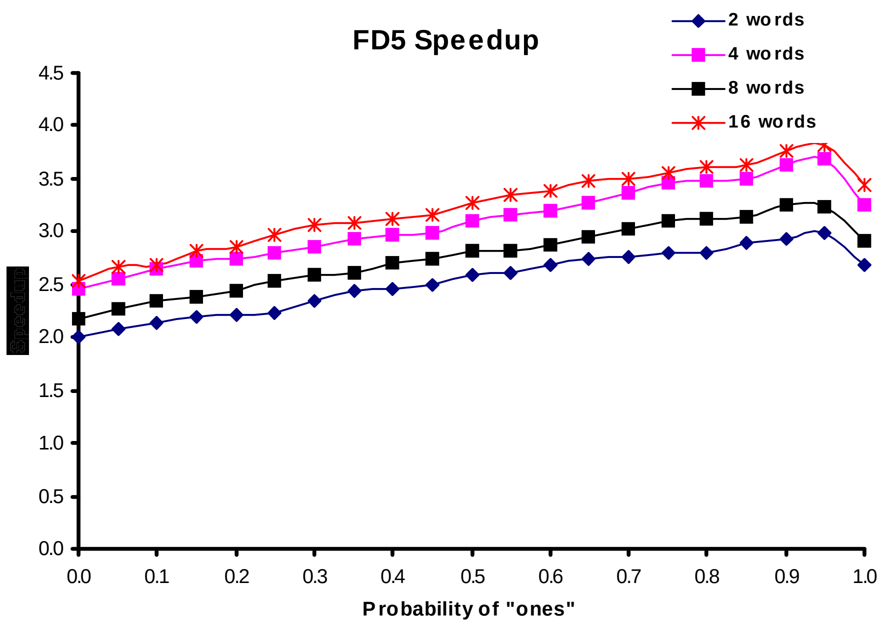


Figure 11: Gained speedup resulted against single word

In Figure 11, if we select a point like 0.05, which means that most of the bits in the binary vector will have, 0’s and the others will have 1’s, then our suggested technique will be faster than the hardware implementation of popcount by a factor of 2.

7. CONCLUSIONS

Popcount is used in various applications such as information retrieval, file processing, and coding theory. In information retrieval, the number of “1”s in the array of binary vectors presets the characteristic function of a set of retrieved values. This can be used to decide whether to constrain or broaden the search criteria to ensure selection of the desired items. The comparison operation between two given files in terms of Hamming distance can also employ popcount operation. Important demands of graphics manipulation have prompted using the popcount in some graphic routines in Windows system (such as routine one ( )).

In this paper, we have presented a software technique for counting the number of “ones” that is faster than the hardware implementation. The speedup varies depending on the nature of the numbers. Further merging popcount hardware implementation with the software implementation will improve the speedup.

The presented results are a product of the machine, operating system, implementation, and compiler design. These factors are related to each other and they might affect our results.

Popcount is an unusual instruction since it would be hard to generate from most programming languages. It is recommended to add this instruction to all programming languages such that it should operate on nonnegative integers. There is absolutely no difficulty in adding this instruction as an intrinsic function, library macro, etc., and/or a compiler generating inline code and using hardware support where, or/and when appropriate (El-Qawasmeh, 2001).

REFERENCES

1- Berkovich, S., Lapir, G., and Mack, M. (2000): A Bit-Counting Algorithm Using the Frequency Division Principle. Software: Practice and Experience 30(14):1531-1540.

2- Berkovich, S., El-Qawasmeh, E., Lapir, G., Mack, M., and Zincke, C. (1998): Organization of Near Matching in Bit Attribute Matrix Applied to Associative Access Methods In Information Retrieval. Proc. of the 16th IASTED International Conference Applied Informatics, Garmisch-Partenkirchen, Germany: 62-64.

3- El-Qawasmeh, E. and Hemidi, I. (2000): Performance Investigation of Hamming Distance Bit Vertical Counter Applied to Associative Access Methods in Information Retrieval. Journal of the American Society for Information Science, USA, 51(5):427-432.

4- El-Qawasmeh, E. (2001): Performance Investigation of Bit-Counting Algorithms With a Speedup to Lookup Table. Journal of Research and Practice in Information Technology, Australia, 32(3/4), pp. 215-230.

5- Gutman, R. (2000): Bit-Parallelism. Dr. Dobb’s Journal 25(9):133-134.

6- Reingold, E., Nievergeit, J. and Deo, N. (1977): Combinatorial Algorithms: Theory and Practice. Englewood Cliffs, New Jersey 07632, Prentice Hall.

7- Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. (1995) Numerical Recipes in C. Second edition. Reprinted with correction, Cambridge University Press.