An Optimal Bit-counting Algorithm

MARILYN MACK[1], GENNADI M. LAPIR[2], SIMON BERKOVICH[3]

[1] National Aeronautics and Space Administration,
Goddard Space Flight Center
Code 585 / Code 933
Greenbelt, MD 20771
marilyn.j.mack@gsfc.nasa.gov

[2] ZERES GmbH Universiteatstrasse 142,
44799 Bochum, Germany,
lapir@zeres.de

[3] Department of Computer Science,
School of Engineering and Applied Science,
George Washington University,
Washington, DC, USA,
berkov@seas.gwu.edu

This paper addresses the omnipresent problem of counting bits - an operation discussed since the very early stages of the establishing of computer science, (see [1]). The need for a quick bit-counting method acquires a special significance with the proliferation of search engines on the Internet. It arises in several other computer applications. This is especially true in information retrieval in which an array of binary vectors is used to represent a characteristic function (CF) of a set of qualified documents. The number of "1"s in the CF equals the cardinality of the set. The process of repeated evaluations of this cardinality is a pivotal point in choosing a rational strategy for deciding whether to constrain or broaden the search criteria to ensure selection of the desired items. Another need for bit-counting occurs when trying to determine the differences between given files, (images or text), in terms of the Hamming distance. An Exclusive OR operation applied to a pair of files results in a binary vector array of mismatches that must be counted.

The proposed procedure is presumably the best among the available software methods of bit-counting. It would retain its value in the future if wide-spread special realization of bit-counting hardware does not become cost-effective. Such a fast bit-counting algorithm could increase the speed at relatively low cost and should have a spillover effect on the overall higher operation's speed.

How did the proposed method evolve? It began with a study of existing approaches. Formally, the bit-counting problem can be presented as follows: given an array of binary vectors, it is necessary to determine how many "1's it contains. Despite the apparent simplicity of the task, this problem is associated with interesting algorithmic issues and important applications. (See [2]) The few basic methods used to date are simple in concept with various degrees of efficiency. They are:

(1) Table Lookup (TL) The counting of the bits is done using a table that contains the pre-calculated quantities of "1"s for each entry. The time for getting this number in one access would be comparable to the time for a direct hardware realization of the task. However, operating with words of 32 bits would require a prohibitively large array of 2^{32} bytes. Thus, though it is fast, Single-fetch Table Lookup (TL1) is not as yet practical. Moreover, it becomes impossible when one considers the near-term prospect of 64 bit words. But considering the given word divided into two or four parts can decrease the size of the look-up table. So, for 32 bit words, the size of the required look-up table can be less, correspondingly, 2^{16} (Double Lookup or TL2) and 2^{8} words (Quadruple Lookup or TL4). However, using either of these methods slows down bit-counting since each requires extra accesses, additions, and shifting operations.

(2) Shifting (SH) The simplest and most natural method of counting "1"s in an n-bit number is the Shifting (SH) method in which each of the bits is effectively examined by shifting, i.e. given a test array, C, of n vectors each of length w, each of the n components would separately count "1"s using w shifts for its length w, for all n vectors. This method is straightforward but its execution time is in direct proportion to nw. As such, it would not be efficient for large sparse arrays.

(3) Algebraic Logic (AL) A faster bit sum algorithm, described in [1] and [2], makes use of the way in which arithmetic is performed on most computers. The procedure repeatedly re-calculates the number as the logical sum, ANDing, of the current value of the number and its decrement by one until the result becomes zero. Thus its timing is in direct proportion to the number of set bits rather than to the full machine word size, (as is the case with SH).

(4) Vertical Counting (VC) The idea of vertical transformations has been realized in a radical development in computer architecture - the STARAN computer of the Goodyear Corporation [3][4]. The VC

method can be considered as an emulation of addition for 32 bit registers. The XOR (Exclusive OR) gate determines corresponding characters that mismatch while AND gates generate carries. VC incorporates these concepts into a software equivalent. Initially, as with the half-adder, the carry bits (the higher level positions) in the vertical vector are zero. The sum value is determined by bit XORing the current value of the vertical counter, v_c[i], with the carry bit for position zero. Then the new value of the next power of two vertical counter vector, v_c[i+1], is determined by bit ANDing the current value of that vector with the calculated carry value from v_c[i] to create both a new value for v_c[i+1] and a new carry value. Repeating the ANDing and ORing through successively higher powers of two until the calculated carry value is zero continues this carry propagation. This method is depicted in Figure 1. The utilization of vertical format in bit-counting as expressed in VC has previously been described in [5].

Typical timing test results for most basic methods, AL, VC, TL1, TL2, and TL4 compared with the ideal (represented by a 32-bit machine software emulation of a perfect hardware implementation) for various percentages of randomly set bits (from 0 through 100% in steps of 10) revealed that, the TL methods were most consistently low (Figure 2). But, as discussed, they were not practical. Clearly a faster method was required. The initial bit counting method, VC, had involved counting the bits and then converting the contents of the Vertical Counter into regular horizontal format. However it was not very fast, O(nlogn). This was due to the penalties incurred in dense situations in which extra work is required to approach zero. If the counting was performed earlier than in VC and was done using the faster AL method rather than the SH approach, and if tests for zero were inserted to satisfy sparse cases, time could clearly be saved. This idea resulted in the Lower-bit Sieve (LS) Algorithm. The suggested LS combines the best characteristics of the VC with those AL. The algorithm operates as follows. First, the incoming binary vector enters the "Sieve" - a Vertical Counter of a fixed *small* number of stages. Suppose this number were 2. This provides 32 2-bit registers that become mod 4 counters. These counters operate as a sieve retaining 2 low-bits and submitting overflows to the next stage. Thus the next stage only has to count the bits in quadruplets. As a result, counting at this stage can take advantage of the faster operation of the AL technique in sparse vectors. Since the number of stages in the sieve is small, it can be presented as a "flat" sequence of instructions rather than a loop. The time penalty for denser numbers in the AL method is removed by providing for the AL counting of the "1"s only in the Vertical Counter array, (which is less than those which had been in the original number), without any significant loss of time for the more sparse cases. In the same way, the slower part of the VC method, the final count, is improved. It should also be noted that studies reveal a tendency for the bits randomly set to one to be evenly distributed as the number of candidate words increases. Thus similar time penalties, either proportional to word length as in SH or to bits set as in AL, associated with counting this denser Vertical-Counter number would ordinarily be incurred. But through by-passing the calculation of all carries beyond some point, and then AL counting the set carry bits represented at that point, density time difficulties would be avoided.

Several tests were made to validate these theories. These tests were to be designated as LSj_k where j was the power of the highest carry which could ever be calculated in this instantiation, and k was the number of inserted zero tests. In these tests, the complete Lower-bit Sieve (LS) family included second through seventh carries, (designated as j=2 through 7), and inserted additional tests for zero to reduce the sparse case processing time, (designated as k=0 through 6). Clearly, all members of this LS family of methods would eliminate time counting all bits and would only count combinations of bits. The overall effect of this combination would result in the fastest method of all.

The presented LS technique, depicted in Figure 3, has a definite advantage for sparse vectors. This means that in the case of dense vectors, it would be beneficial to count "0" bits rather than "1" bits. In general, when the bit-distribution of binary vectors is not known, the utilization of this circumstance is uncertain. However, in the case of information retrieval, certain patterns in bit-distribution have been observed. Namely, most of the searching requests result either in "success" - small number of retrieved items or in a broad definition - exceedingly many retrieved items. This suggests the following strategy for the evaluation of the characteristic vectors: to count "1"s and "0"s simultaneously. Counting "0" is done by logically ANDing the number and its increment until the result is all "1"s and then subtracting the determined count from the bit-length of the array. The first count process to finish terminates the other. As both processes run simultaneously, they can benefit from intrinsic instruction level parallelism inherent to modern microprocessors. The actual results of these simulations were consistent with predicted results. It turns out that up to the sparsity of about 15%, the combined method is faster and thus can be recommended for information retrieval systems.

The results of completed experiments for various implementations of each level in the family of LS methods were normalized relative to the ideal (Figure

4). These tests confirmed the strengths of the algorithm by showing a close simulation in time for the assumed fastest "ideal" method. LS6_4, for example, required approximately 1.8 times the speed of ideal while AL had required 9.7 times that same value. Though no one LS level was significantly higher for all possible distributions of ones tested individually, what emerged was the fact that the average dispersion of ones for a particular application, i.e. whether the vectors were sparse, often determined which one of the various LS methods tested was the best. As such, the choice is application specific. Additional tests on a native 64 bit CRAY machine confirmed the algorithm's independence from word size.

Why does this proposed procedure perform so well? The algorithm is close to hardware in spirit. It incorporates principles of vertical approaches with Instruction Level Parallelism (ILP). It also compares well with the ideal implementations of bit-counting in super-computers as represented by the HPF function popcnt(). Thus the current speed gain due to the suggested Lower-Bit Sieve family of methods would enable an efficient automatic retrieval of qualifying records through automatic, fast tolerance threshold adjustment. Such criteria modification, performed without the need to engage the user, could quickly and automatically tighten restrictions when too vast an amount of data qualified and loosen them if little or no data was secured. So tolerable, user-friendly access to information would be enabled while accuracy and speed are maintained within reasonable limits.

In conclusion, this paper presents a faster approach to bit-counting that should, through spillover effects, improve overall performance of higher level operations in which it is used.

REFERENCES

[1] Derrick H. Lehmer, The Machine Tools of Combinatorics, in *Applied Combinatorial Mathematics*, Edwin F. Beckenbach [Ed.], (New York, NY: John Wiley and Sons, Inc., 1964).

[2] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo, *Combinatorial Algorithms Theory and Practice*, (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977).

[3] Kenneth J. Thurber and Leon D. Wald, Associative and Parallel Processors, *Computing Surveys*, (New York, NY: ACM, vol. 7, No. 4, December, 1975, pp. 215-255).

[4] Kenneth E. Batcher, The Multidimensional Access Memory in STARAN, *IEEE Transactions on Computers*, (New York, NY: IEEE, vol. C-26, No. 2, February, 1977, pp. 174-177).

[5] Simon Berkovich, Eyas El-Qawasameh, Gennadi M. Lapir, Marilyn Mack, Christopher Zincke, "Organization of Near Matching in Bit Attribute Matrix Applied to Associative Access Methods in Information Retrieval", *16th IASTEDInternational Conference on Applied Informatics*, (Garmisch-Partenkirchen: IASTED, February 23-25, 1998).

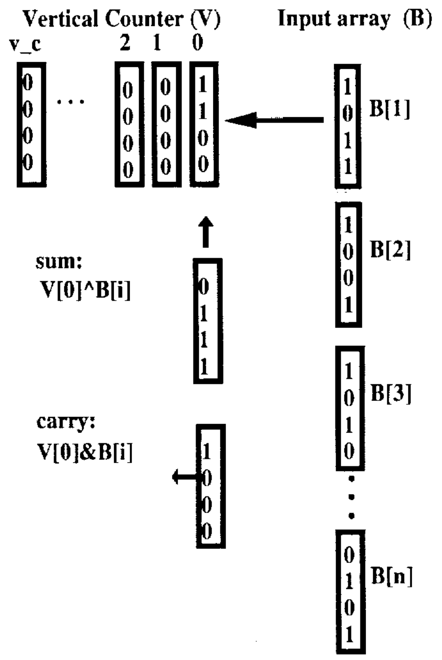

Figure 1: Organization of Vertical Counting (VC)

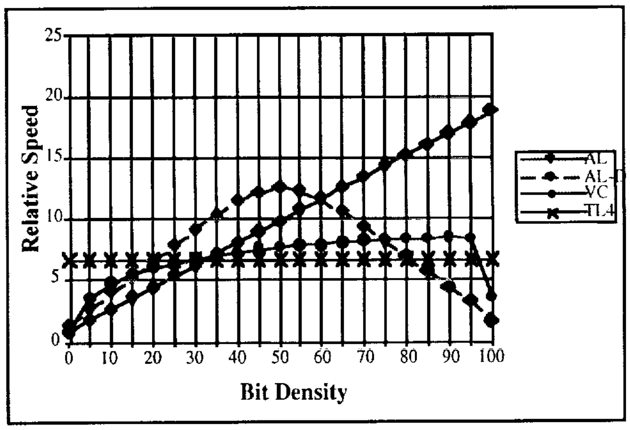

Figure 2: Performance of the basic bit counting techniques (normalized to the ideal case: emulated popcnt)

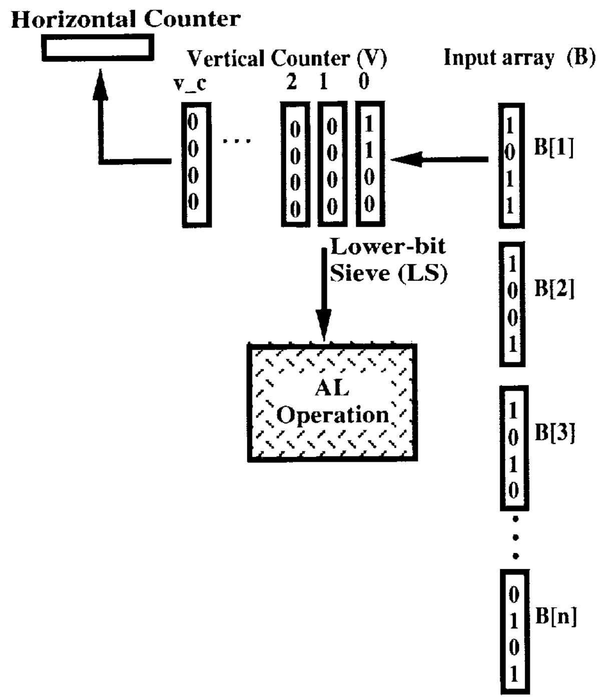

Figure 3: Model of Lower bit Sieve (LS) - Options for Vertical Counting method (upper arrow counting all values) and Lower bit Sieve Method (lower arrow counting values more efficiently)

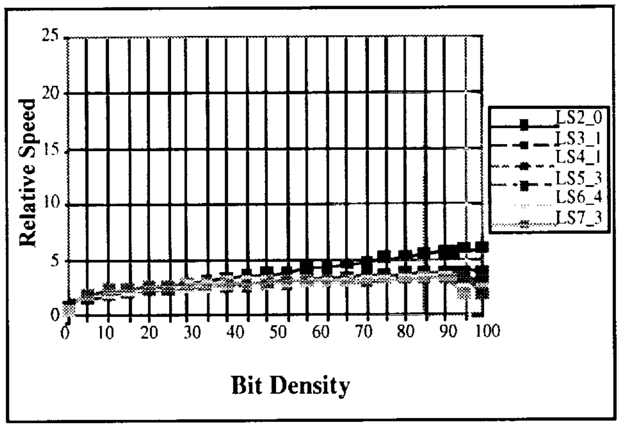

Figure 4: Performance of the family of Lower bit Sieve algorithms (normalized to the ideal case: emulated popcnt)