Population count in arrays

Y. Edel and A. Klein

March 6, 2009

Abstract

We give a new algorithm for the population count on columns of a bit-matrix. Even as ordinary population count algorithm, if applied to large arrays, it performs better than the older algorithms.

1 Introduction

The function that returns the number of 1-bits of a binary word is called *population count* or *side way addition*. We will call it `popc` for short.

It is an important task that occurs in several contexts. Some computers have even a special machine instruction for it, e.g. it is called `SADD` (on a Mark I), `POPCNT` (in the SSE4.2 instruction set) or `VPCNT` (on a NEC SX-4).

Since many current architectures (like the Intel i386) do not have a machine level population count, algorithms that compute the population count with simple logic and arithmetic operations are still important.

In this paper we give an algorithm for counting the 1-bits in the columns of a bit-matrix, i.e. for the "vertical population count" problem (see Section 3), which arises in some contexts. Such an algorithm is even on machines with build in population count instruction of importance, as it is expensive to transpose a binary matrix.

Also as usual population count algorithm it is faster than the older algorithms if the array is large (some 1000 words).

Some applications of the population count are:

- Generation of random variables with binomial distribution by applying popc to a random variable with uniform distribution (as suggest by J.H. Ahrens, see exercise 3.4.1.27 in [5]).
- The Hamming distance between x and y is $\mathrm{popc}(x \oplus y)$.
- Several fast correlation attacks on LFSR based stream ciphers [4] need to evaluate the population count for large arrays and have to count the number of 1-bits in the columns of a bit-matrix.

The last application was the reason for us to look at this problem.

2 Known algorithms

In this section we summarize the most important population count algorithms. We only consider general purpose algorithms. There are also specialised Algorithms for spare populated words (see Wegner's Algorithm [10, 6, 9]).

In the algorithms frequently occur the constants μ_k as masks. We define μ_k as the number whose 2-adic representation is the repeated sequence of 2^i ones followed by 2^i zeros.

So if we denote by $(\ldots, x_2, x_1, x_0)_b$ the integer with b-adic representation $x_0 + x_1 b + x_2 b^2 + \ldots$, we have:

$$\begin{aligned} \mu_0 &= (\ldots 0101010101010101)_2 \\ \mu_1 &= (\ldots 0011001100110011)_2 \\ \mu_2 &= (\ldots 0000111100001111)_2 \\ &\vdots \end{aligned}$$

For real world comparison we have implemented the algorithms C [3]. To avoid introducing typos we give all algorithms as C listings. As we do not use very special C features the reader should be able to understand them even without knowledge of the C programming language. We only point out that:

- a&b denotes the bitwise-`AND` of a and b.
- **uint32_t** and **uint64_t** denote unsigned integers of **exactly** 32 or 64 bits respectively.

All other type declarations are not important.

2.1 Table lookup

A simple way to do the population count is to store the 256 values $[\text{popc}(0), \ldots, \text{popc}(255)]$ in a table and do the computation by a table lookup for every byte. This algorithm is not particular deep, but very efficient on 32-bit machines. (The boost project [1] uses it for the `dynamic_bitset` class).

2.2 Divide and Conquer Algorithms

In a divide and conquer algorithm we start (for example) with a 32-bit word $(a_{31}, \ldots, a_0)_2$. In the first step we compute $b = (b_{15}, \ldots, b_0)_4$ with $b_i = a_{2i} + a_{2i+1}$. Next we compute $c = (c_7, \ldots, b_0)_8$ with $c_i = b_{2i} + b_{2i+1}$ and repeat this until we reach base 64. Listing 1 shows the corresponding C code.

Such an algorithm appears already in the fist textbook on programming [11] for the 35-bit EDSAC.

There are many ways to optimize the basic divide and conquer algorithm. Listing 2 shows a good variant for a 64-bit machine which has a fast multiplication.

Listing 1: A simple divide and conquer algorithm (32-bit)

```
1 uint_fast8_t popc_unoptimized(uint32_t x) {
2   x = (x & mu0) + ((x>>1)  & mu0);
3   x = (x & mu1) + ((x>>2)  & mu1);
4   x = (x & mu2) + ((x>>4)  & mu2);
5   x = (x & mu3) + ((x>>8)  & mu3);
6   x = (x & mu4) + ((x>>16) & mu4);
7   return x;
8 }
```

Listing 2: An optimized divide and conquer algorithm (64-bit)

```
1 uint_fast8_t popc_mul(uint64_t x) {
2   x = x - ((x>>1) & mu0);
3   x = (x & mu1) + ((x>>2) & mu1);
4   x = (x + (x>>4)) & mu2;
5   const uint64_t M = UINT64_C(0x0101010101010101);
6   return ((M*x) >> 56);
7 }
```

Apart from using the fast multiplication line 2 is also altered. What happens here can be best understood if we look at 2-bit words. For $x = (x_1x_0)_2$ the operation x − ((x>>1)&mu0) gives $(x_1x_0)_2 - (0\,x_1)_2 = 2x_1 + x_0 - x_1 = x_0 + x_1$. So we still compute the base 4 sum and save one `AND`-instruction.

On 64-bit machines divide and conquer algorithms usually beat the table lookup algorithm.

2.3 The Harley-Seal method

To our knowledge the, up to now, fastest algorithm to compute the population count for large arrays was an algorithm developed by Harley and later improved by Seal [7], which is based on a carry save adder (CSA).

A CSA (or 3 : 2 compressor) is just a sequence of independent full adders. A full adder is a circuit respectively a function that takes three bits a, b, c and computes the two bit sum $a + b + c$.

The essential equation for the Harley-Seal method is:

$$\mathrm{popc}(a)+\mathrm{popc}(b)+\mathrm{popc}(c) = 2\,\mathrm{popc}(h)+\mathrm{popc}(l) \qquad \text{with } (h,l) = CSA(a,b,c).$$

Listing 3 shows how Seal use the CSA to compute the population count of an array.

The idea behind the program of Listing 3 can be iterated. With each iteration the complexity of the code increases. After some iterations we reach a

Listing 3: Harley-Seal Algorithm for population count

```
1  uint_fast32_t popc_HS(word* a) {
2    uint_fast32_t total=0;
3    word ones=0,twos;
4    int_fast32_t i=0;
5    for(; i<=N-2;i+=2) {
6      CSA(&twos,&ones,ones,a[i],a[i+1]);
7      total += POPC(twos);
8    }
9    total = 2*total + POPC(ones);
10   for(; i<N;++i) {
11     total += POPC(a[i]);
12   }
13   return total;
14 }
```

point where the additional cost for loads etc. dominate the benefit. It seams that the third iteration as shown in Listing 4 is a good compromise. Further iterations give either only marginal improvements or even hurt.

In the revisions for his book [8] Warren describes and analyses these algorithms in detail. If the population count need n_p operations and the carry save adder needs n_c operations the k-th iteration of the Harley-Seal Method needs $n_c + \frac{n_p - n_c + 1}{2^k}$ operations per word (ignoring loads and loop control). For $n_c = 5$ and $n_p = 15$ we obtain that the program Listing 4 needs 6.38 instructions per word. This is a speedup of 60% in comparison to the 16 instruction used in the naive method which simply sums up the population count for all words. Experiments (see Section 5) on real machines show that, due to the complexer structure of the program, the speedup is only 40% which is still impressive.

2.4 Bit splicing algorithms

By the name bit splicing algorithms we refer to algorithms that base on the idea that a 64-bit adder can viewed as k parallel $64/k$-bit adders as long as no carry is involved. For example the divide and conquer algorithm of section 2.2 is an straight forward application of bit splicing.

An example that makes better use of bit splicing is the algorithm given by H. S. Warren, Jr. in his book [9] (see Listing 5). (`word` denotes either an unsigned 32- or 64-bit integer type depending on the machine we use.)

As in listing 2 the lines $7-9$ produce a number $x = (x_7, \ldots, x_0)_{256}$ with the additional property $0 \le x_i \le 8$. Since $8 \cdot 31 = 248 < 255$ we can sum 31 of these numbers without a carry (to base 256).

This algorithm is about 20% faster on large arrays than the naive population count algorithm.

Listing 4: iterated Harley-Seal Method

```
1  uint_fast32_t popc_HS3(word* a) {
2    uint_fast32_t total=0;
3    word ones=0,twos=0,fours=0,eights;
4    word twosA,twosB,foursA,foursB;
5    int_fast32_t i=0;
6    for(; i<=N-8;i+=8) {
7      CSA(&twosA ,&ones,ones,a[i  ],a[i+1]);
8      CSA(&twosB ,&ones,ones,a[i+2],a[i+3]);
9      CSA(&foursA,&twos,twos,twosA,twosB);
10     CSA(&twosA ,&ones,ones,a[i+4],a[i+5]);
11     CSA(&twosB ,&ones,ones,a[i+6],a[i+7]);
12     CSA(&foursB,&twos,twos,twosA,twosB);
13     CSA(&eights,&fours,fours,foursA,foursB);
14     total += POPC(eights);
15   }
16   total = 8*total +
         4*POPC(fours)+2*POPC(twos)+POPC(ones);
17   for(; i<N;++i) {
18     total += POPC(a[i]);
19   }
20   return total;
21 }
```

In the revisions for his book Warren abandons this algorithm in favor to faster Harley-Seal method.

Our new algorithm is also based on the "bit splicing" idea. The problem with Algorithm 5 and other bit slicing algorithms (see [2]) is the following:

31 numbers in the range $[0, 8]$ are summed up thus the result lies in the range $[0, 248]$, but a byte can store numbers up to 255.

So we do not make full use of the 4 (or 8) "parallel 8-bit adders". I.e. the parallel adder is used only 31 times where we would like to use it 31.875 times.

The difference may look marginal, but our algorithm shows that a careful design of a "bit splicing" algorithm, which avoids such a waste, is competitive to the fastest Harley-Seal variant, while in contrast other bit splicing algorithms are much slower.

3 Vertical population count

When we developed our method we were not directly interested in a population count algorithm. We looked at following related problem:

Given a $k \times n$ matrix $A = (a_{i,j})$ of bits. The rows of the matrix are stored as bit-fields and the matrix itself is an array of bit-fields. How can the population

Listing 5: Warren's Algorithm for population count in an array

```
1  uint_fast32_t popc_warren(word a[restrict]) {
2    uint_fast32_t res=0;
3    for(uint_fast32_t i=0; i<N; i+=31) {
4      uint_fast32_t lim = min(N, i+31);
5      word sumbytes =0;
6      for(uint32_t j=i;j<lim; ++j) {
7        word x = a[j] - ((a[j]>>1) & mu0);
8        x = (x & mu1) + ((x>>2) & mu1);
9        x = (x + (x >> 4)) & mu2;
10       sumbytes +=x;
11     }
12
13     uint_fast8_t *p = (uint_fast8_t*)(&sumbytes);
14     for(uint_fast8_t x=0; x<sizeof(word); ++x) {
15       res += p[x];
16     }
17
18   }
19   return res;
20 }
```

count for each column of the matrix be done? (I.e. we want to compute the numbers $p_j = \sum_{i=0}^{k-1} a_{i,j}$ for each $j = 0, \ldots, n-1$.)

The straight forward approach to solve this problem is to transpose the matrix A and apply the classical population count algorithms to the n bit fields in A^t.

This approach leads not to an effective algorithm, since there is no really fast way to transpose a bit matrix (see [9] Section 7.3). The, to our knowledge, the fastest way to transpose a bit matrix on Intel hardware uses the `pmovmskb` instruction.

This instruction interprets a 64-bit word as 8×8 matrix and returns the byte standing in the first column. By repeatedly shifting the 64-bit word to the left and applying the `pmovmskb` instruction we can get all columns of the 8×8 matrix as bytes.

But even transposing a bit matrix this way needs many operations. So unless our machine has a machine level population count instruction and a generalised `pmovmskb` instruction that computes the full transpose instead of only the first column (see also the comments on `pmovmskb` in [12]) an other method to do the vertical population count is needed.

Our solution works on a 255×64 matrix (or 255×32, if we have a 32-bit machine). It utilizes the "bit splicing" idea, but never wastes potential parallel additions.

Listing 6: Vertical population count for a 255×64 matrix

```
1  void popc_255(const word a[const 255],word d[8]) {
2    static word b[170];
3    for(uint_fast8_t i=0; i<85; ++i) {
4      b[2*i] = (a[3*i]&mu0) + (a[3*i+1]&mu0) +
           (a[3*i+2]&mu0);
5      b[2*i+1] = ((a[3*i]>>1)&mu0) + ((a[3*i+1]>>1)&mu0) +
           ((a[3*i+2]>>1)&mu0);
6    }
7
8    static word c[68];
9    for(uint_fast8_t i=0; i<17; ++i) {
10     for(uint_fast8_t x=0; x<2; ++x) {
11       c[4*i+x] = b[10*i+x]&mu1;
12       c[4*i+2+x] = (b[10*i+x]>>2)&mu1;
13       for(uint_fast8_t j = 2;j<10;j+=2) {
14         c[4*i+x] += b[10*i+j+x]&mu1;
15         c[4*i+2+x] += (b[10*i+j+x]>>2)&mu1;
16       }
17     }
18   }
19
20   for(uint_fast8_t x=0; x<4; ++x) {
21     d[x] = c[x]&mu2;
22     d[4+x] = (c[x]>>4) & mu2;
23     for(uint_fast8_t j = 4;j<68;j+=4) {
24       d[x] += c[x+j]&mu2;
25       d[4+x] += (c[x+j]>>4) & mu2;
26     }
27   }
28
29   return;
30 }
```

First we split a word $x = (x_{63}, \ldots, x_0)_2$ in two words $x' = (x_{62}, x_{60}, \ldots, x_2, x_0)_4$, $x'' = (x_{63}, x_{61}, \ldots, x_1)_4$ using the mask μ_0 and shifts.

Then, for three words we add the corresponding splited words x', y', z' resp. x'', y'', z''. Since $x_i, y_i, z_i \leq 1$ we can add x', y', z' resp. x'', y'', z'' without having a carry in base 4.

Note that $x_i + y_i + z_i$ can be 3, which is the maximum that can be done in base 4 without carry and so the "parallel 2-bit adders" have been used as much as possible.

In the next step we proceed similarly with the so obtained base 4 words.

We split a base 4 word b in two base 16 words b', b''. We take 5 base 4 words $b_0, b_1, ..., b_4$ and add the corresponding splited words. Since the digits in base four are less or equal to 3 and $3 \cdot 5 = 15 < 16$ the sums $b'_0 + b'_1 + b'_2 + b'_3 + b'_4$ and $b''_0 + b''_1 + b''_2 + b''_3 + b''_4$ cab be computed without a carry in base 16. Again note that the sums can reach 15, i.e. the "parallel 4-bit adders" are used as much as possible.

In the final step we do the analog procedure on the base 16 words and go from base 16 to base 256. Since $15 \cdot 17 = 255 < 256$ we can sum 17 words in the final step and again make optimal use of the "parallel 8-bit adders".

Base 256 means, that we have reached bytes, which can be addressed directly and so we stop at this point. All together we need $3 \cdot 5 \cdot 17 = 255$ words as input for our algorithm and get as output 8 words (on a 64-bit machine, that are $8 \cdot 8 = 64$ bytes) which contain the result of the vertical population count.

Listing 6 shows a C implementation of our algorithm. The array `d` contains the 8 words with the vertical population counts. We omit the conversion of the array of words to an array of bytes. It is a bit difficult to do this in portable code, since we must pay attention to little and big endian architectures to address the byte corresponding to a given column.

The code of Listing 6 is very efficient. If we simply add the 64 bytes at the end, we already have an attractive algorithm for the generic population count (on an array of 255 words). In this case we do not have to pay attention if we are working on little or big endian architectures. Even in this basic form it is competitive to the Harley-Seal method.

4 Optimizing the algorithm for the generic population count

If we only want to sum all bits and are not interested in the vertical population count, the speed of our new population count algorithm can be slightly improved.

Out first modification is to replace the lines 4 and 5 of Listing 6 by:

```
4  b[2*i] = (a[3*i] - ((a[3*i]>>1)&mu0)) + (a[3*i+2]&mu0);
5  b[2*i+1] = (a[3*i+1] - ((a[3*i+1]>>1)&mu0)) +
       ((a[3*i+2]>>1)&mu0);
```

The idea was already explained for algorithm 2. The bits are summed up in a different order compared to Listing 6, but as we are now only interested in the total sum over all bits the order of the additions is not important. This saves one `AND` instruction per line.

In the lines 9 to 18 of Listing 6 we had to take care that we combine the right elements of the vector b. This result in the complicate double loop over i and x. If we do not need the sum of the different columns, the order in which we add the bs can be changed. This leads to the simpler loop:

```
 9  for(uint_fast8_t i=0; i<34; ++i) {
10    c[2*i]   = b[5*i]&mu1;
11    c[2*i+1] = (b[5*i]>>2)&mu1;
12    for(uint_fast8_t j = 1;j<5;++j) {
13      c[2*i] += b[5*i+j]&mu1;
14      c[2*i+1] += (b[5*i+j]>>2)&mu1;
15    }
16  }
```

Finally we can combine the idea of Harley and Seal with our algorithm (this is also possible for the vertical population count variant). The call of the carry safe adder should be inserted after line 3 of Listing 6. In Listing 7 we show the second iteration of the Harley-Seal idea combined with our algorithm, which computes the population count of an array of 1020 words.

We formulate the analysis of our algorithm as:

Theorem 1
With the improvements given above, our algorithm needs 2112 *operations to compute the population count of* 255 *words (not counting loop control, loads, etc.). This are approximately* 8.28 *operations per word.*

If we use in addition i CSA iterations, our algorithm needs $255(2^i-1)n_c + i(n_p+2) + 2112$ *operations to compute the population count of* $2^i \cdot 255$ *words (not counting loop control, loads, etc.), where n_c and n_p denote the numbers of operations needed for CSA or word population count, respectively.*

Proof
With the modification given above we need 5 operations for line 4 and 6 operations for line 5. This makes all together $85 \cdot (5+6) = 935$ operations to compute the b's from the a's. To get the c's from the b's we need two `AND`'s and one `SHIFT` per b and four additions per c, i.e. $170 \cdot 3 + 68 \cdot 4 = 782$ operations.

To compute the eight d's we need $68 \cdot 3 + 8 \cdot 16 = 332$ operations. For the final step of summation the 64 bytes contained in the d's we need 63 operations.

All together these are 2112 operations.

If we use in addition i CSA iterations we have additionally inside the first loop $3(2^i - 1)$ CSA calls, and in the final combining step i word-population counts, i additions and i multiplications.

So we have in total $255(2^i-1)n_c + i(n_p+2) + 2112$ additional operations and processed $2^i \cdot 255$ words. □

If we choose the version with i CSA iterations, $i = 1, 2, 3$ and use, like Warren, $n_c = 5$ and $n_p = 15$ we see that our algorithm needs about 6.67, 5.85 resp. 5.43 operations per word. For comparison the third iteration of the original Harley Seal algorithm (see algorithm 4) needs about 6.38 operations per word.

Listing 7: population count in an array of size 1020

```
1   uint_fast16_t popc_1020(word* a) {
2   static word b[170];
3   word one=0,two=0,twoA,twoB;
4   for(uint_fast8_t i=0; i<85; ++i) {
5     word X,Y,Z;
6     CSA(&twoA,&one,one,a[12*i   ],a[12*i+1]);
7     CSA(&twoB,&one,one,a[12*i+2],a[12*i+3]);
8     CSA(&X,&two,two,twoA,twoB);
9     CSA(&twoA,&one,one,a[12*i+4],a[12*i+5]);
10    CSA(&twoB,&one,one,a[12*i+6],a[12*i+7]);
11    CSA(&Y,&two,two,twoA,twoB);
12    CSA(&twoA,&one,one,a[12*i+8],a[12*i+9]);
13    CSA(&twoB,&one,one,a[12*i+10],a[12*i+11]);
14    CSA(&Z,&two,two,twoA,twoB);
15    b[2*i   ] = (X - ((X>>1)&mu0)) + (Z&mu0);
16    b[2*i+1] = (Y - ((Y>>1)&mu0)) + ((Z>>1)&mu0);
17  }
18  static word c[68];
19  for(uint_fast8_t i=0; i<34; ++i) {
20    c[2*i]   = b[5*i]&mu1;
21    c[2*i+1] = (b[5*i]>>2)&mu1;
22    for(uint_fast8_t j = 1;j<5;++j) {
23      c[2*i] += b[5*i+j]&mu1;
24      c[2*i+1] += (b[5*i+j]>>2)&mu1;
25    }
26  }
27  static word d[8];
28  for(uint_fast8_t x=0; x<4; ++x) {
29    d[x] = c[x]&mu2;
30    d[4+x] = (c[x]>>4) & mu2;
31    for(uint_fast8_t j = 4;j<68;j+=4) {
32      d[x] += c[x+j]&mu2;
33      d[4+x] += (c[x+j]>>4) & mu2;
34    }
35  }
36  uint_fast8_t *p = (uint_fast8_t*)d;
37  uint_fast16_t res =p[0];
38  for(uint_fast8_t x=1; x<8*sizeof(word); ++x) {
39    res += p[x];
40  }
41  return 4*res+2*POPC(two)+POPC(one);
42 }
```

Experiments show that in practice the algorithms do not benefit from more than three iterations due to housekeeping effects (loops, control, memory, ...). For 2-3 iterations, the real world results are close to the theoretical estimates.

Our algorithm uses complex loops and many variables. It can benefit from a good loop unroll and an optimization of the memory access. For example it is possible to rewrite the algorithm, such that it has no large local arrays. The automatic optimizer of development version of the GNU C compiler (the upcoming version 4.4) does this quite well, for older versions of the gcc, we use variants of the code in which we manually unrolled the loop. We remark that manually unrolled loops can easily disturb the optimizer of the compiler, so such transformations should be used carefully.

An other possible optimization, on a Intel maschine, is to use 128-bit SSE registers and instructions. This works for our algorithm as well as for the divide and conquer algorithms and the Harley-Seal method. Of course the code is no longer portable and runs only on Intel machines with SSE support. In our tests we were able to obtain an gain of almost 50% compared to the 64-bit code.

5 Timings

As already mentioned, it is difficult to predict the exact run time, as the number of registers, cache size, the time needed for things like loop control or memory access differs considerably between different machines. The following tables show timings for 32 and 64 bit hardware. Note that in a real application other parts of the program can influence the cache, so the timings should be consider as approximate values.

Algorithm	Timing
Simple loop (table lookup)	100%
Warren's bit splicing algorithm	92%
Harley-Seal	73%
third iteration of Harley-Seal	59%
vertical population count	71%
optimised vertical population count	52%
Listing 7	39%

Table 1: Timings for a 32-bit Intel Core2 CPU under Linux (gcc 4.4)

References

[1] Boost. http://www.boost.org.

[2] Testing the bitslice algorithm for popcount. `http://www.dalkescientific.com/writings/diary/archive/2008/07/05/bitslice_and_popcount.html`.

Algorithm	Timing
Simple loop (Listing 2)	100%
Warren's bit splicing algorithm	100%
Harley-Seal	75%
third iteration of Harley-Seal	56%
optimised vertical population count	53%
Listing 7	40%

Table 2: Timings for a 64-bit Quad Core Intel Xenon under Mac OS (gcc 4.4)

[3] Y. Edel and A. Klein. Implementation of population count algorithms. `http://cage.ugent.be/~klein/popc.html`.

[4] T. Johansson and J. J. Jönsson. Improved fast correlation attacks on stream ciphers via convolutional codes. In *Advaces in Cryptology – EUROCRYPT'99*, volume 1592 of *LNCS*, pages 347–362, Berlin, 1999. Springer-Verlag.

[5] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 3 edition, 1998.

[6] D. E. Knuth. *Pre-Fascicle 1a (Bitwise tricks and techniques)*. Addison-Wesley, to apear. available for alpha-testing: `http://www-cs-faculty.stanford.edu/~knuth/fasc1a.ps.gz`.

[7] D. Seal. Newsgroup comp.arch.arithmetic, May 13 1997.

[8] Hompage of Hacker's Delight. `http://www.hackersdelight.org/`. Contains example programs, eratta and additional material.

[9] H. S. Warren, Jr. *Hacker's Delight*. Addison-Wesley, Boston, 2003. Revisions and additional material are on the Hompage of the book [8].

[10] P. Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3:322, 1960.

[11] M. V. Wilkes, D. J. Wheeler, and S. Gill. *The Preparation of Programs for an Electronic Digital Computer*. Addison-Wessley, 2 edition, 1957.

[12] X86 assembly instructions you always wanted but intel didn't give them to you. `http://guru.multimedia.cx/x86-assembly-instructions-you-always-wanted-but-intel-didnt-give-them-to-you/`, September 2006.