Development and Performance Evaluation of Fast Combinatorial Unranking Implementations

András Majdán*, Gábor Rétvári, and János Tapolcai

Department of Telecommunications and Media Informatics
Budapest University of Technology and Economics
{majdan,retvari,tapolcai}@tmit.bme.hu

Abstract. The goal of this paper is to provide fast combinatorial unranking implementations for use in bitmap data compression. Within this context, unranking refers to the operation of obtaining a bit vector given its rank in a particular enumeration of all bit vectors of the same size with respect to a given order. Easily, the simplest way to accomplish this task is to use a lookup table. However, for large block sizes such a table may not fit into the memory. Efficient combinatorial unranking algorithms, which eliminate large lookup tables, are therefore essential in practice. Taking the textbook combinatorial unranking schemes, in this paper we develop very fast combinatorial unranking implementations and we introduce a comprehensive performance evaluation and profiling toolkit to measure their efficiency. Our benchmarks show that our optimized implementations improve the performance of the naive combinatorial unranking implementations by 39%, almost attaining the performance of simple lookup tables.

1 Introduction

Bit vectors, or *bitmaps*, are amongst the most frequently used data structures in the theory and practice of computer science and, more closely, in the networking area. Applications range from storing allocation tables in virtual memory and virtual file systems, raster images in computer graphics or, more generally, storing arbitrary sets over a fixed (large) universe. Today's massive applications, big data analytics softwares, and large-scale network services, however, require fast and efficient operations on enormously large bitmap instances, significantly increasing the computational burden implied by this simplistic data structure.

Storing a bitmap is trivial if memory cost is not to be considered. However, in practice we are often obliged to save space by using bitmap compression algorithms, because the uncompressed bitmap image would simply not fit into the main memory. Recently, bitmap compression algorithms have been demonstrated to effectively reduce the memory footprint of bitmaps without significant performance penalty on standard operations, like random access[1]. This holds much

* The author was supported by the Google Faculty Research Awards programme "Searchable Compressed Data Structures for the Mainstream".

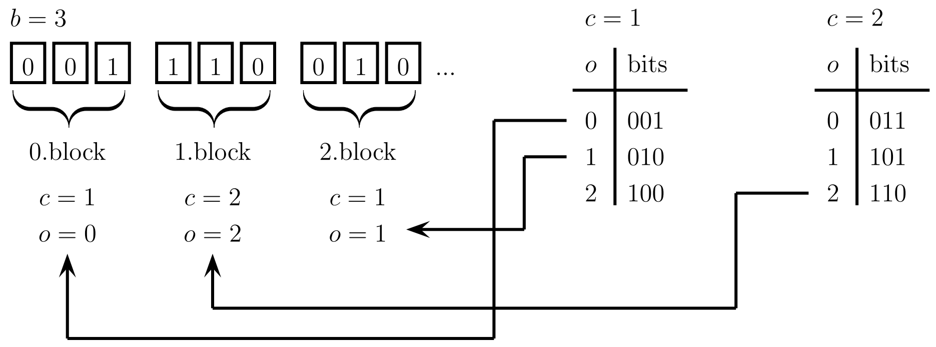


Fig. 1: Bitmap Block Compression Scheme.

promise to application developers, as a way out of the compelling scalability issues caused by today's massive input data sets. For instance, lately compressed bitmaps have been used to squeeze Internet forwarding tables, comprising hundreds of thousands IP address prefix to next-hop associations, to just a couple of hundred kilobytes of memory without any decrease in lookup performance[2].

Consider the following simple bitmap compression algorithm, called the Bitmap Block Compression Scheme (BBCS) (see Fig. 1):

1. Divide a bitmap with size 'n' into blocks of size 'b'.
2. Count the number of 1s in each block and assign this to the block as its class (c).
3. Index the block in a fixed enumeration of all bit vectors of the same class and call it the offset (o).
4. Store the class and the offset of each block sequentially.

Curiously, already this simple block compression scheme can reduce the space requirement of bitmaps to the theoretical minimum (up to lower order error terms). Consider the following result from [8].

Proposition 1. *Given a bitmap of size n and zero-order entropy H_0, the size of the representation compressed by he BBCS algorithm is at most $nH_0 + o(n)$ bits.*

Proof. Block i of the bitmap is represented by the pair (c_i, o_i), where c_i is the number of 1s in the i-th block and o_i is the id of the block within class c_i. Observe that the offset of the i-th block o_i can take $\binom{b}{c_i}$ different values, and hence each o_i can be stored on $\log \binom{b}{c_i}$ bits. Adding this up, for storing the offsets we need

$$\sum_{i=1}^{\lceil \frac{n}{b} \rceil} \binom{b}{c_i} \leq \log \binom{b(n/b)}{c_1 + ... + c_{n/b}} \leq nH_0$$

bits, using the fact that $\log\binom{b}{c_1} + \log\binom{b}{c_2} \le \log\binom{2b}{c_1+c_2}$. The class ids c_i are stored on $\log(b+1)$ bits each, which add up to

$$\sum_{i=1}^{\lceil \frac{n}{b} \rceil} \log(b+1) \le (n+1)\frac{\log(b+1)}{b} = o(n) \ .$$

Thus, we need at most $nH_0 + o(n)$ bits to store the bitmap, which finishes the proof. □

Easily, the key to the BBCS scheme and many more sophisticated bitmap compression schemes based on the same block compression principle, like RRR [1], is step 3. Here, the task is to compute the offset of a given block of 1s and zeros in some fixed enumeration of all bit vectors of the same class. This task is called *combinatorial ranking* [3, 5, 6]. The reverse operation, that is, the reconstruction of a bitmap block given its class and rank within the enumeration, is called *combinatorial unranking.*

As compression algorithms are usually asymetric, with more emphasis on decompression speed than on compression, combinatorial unranking is the operation with more stringent performance requirements. Hence, in this paper we will focus on fast and efficient combinatorial unranking schemes.

In some cases it is not required to decompress a full block but only a position within. For example, consider a bitmap for a memory allocation table, where each bit represents whether or not a particular memory page is allocated. If we would like to know if a page is free we just have to get the corresponding bit in the bitmap. We shall see later that if we need to only partially reconstruct bitmaps we can get better performance with combinatorial unranking algorithms.

The simplest way for combinatorial unranking is using a lookup table (like in Fig. 1). This, as we shall see, is currently the fastest option available. Lookup tables, however, need storing vast bitmap-to-rank mappings in main memory, ruining cache-friendliness of bitmap compression. Depending on the block size b, the whole lookup table might not even fit into the memory (observe that the size of the table needed is $O(2^b)$ bits). Combinatorial ranking/unranking eliminate tables all together, by providing the bitmap-to-offset mapping *algorithmically.* This, however, may cause substantial computational overhead.

The main questions we ask in this paper are *(i)* whether the improved cache-friendliness of combinatorial unranking compensates for the computational overhead as compared to lookup tables and *(ii)* to what extent the overhead can be reduced by optimized implementations. To answer these questions, we have developed very fast combinatorial unranking implementations for use in bitmap decompression, we created a comprehensive performance evaluation and profiling toolkit to compare these implementations to each other, and we used this tool to benchmark our implementations in extensive performance evaluations.

The rest of this paper is structured as follows. In Section 2, we introduce the theoretical background, in Section 3 we present our implementations and optimization strategies, in Section 4 we briefly discuss the performance evaluation

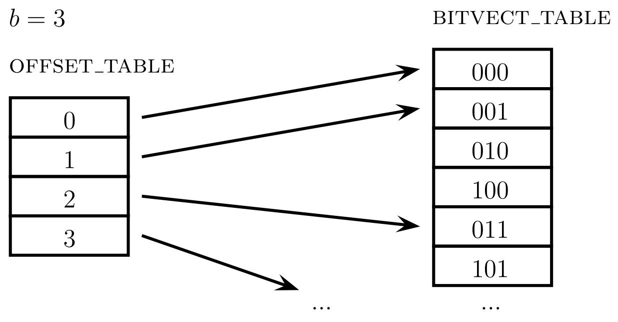


Fig. 2: Lookup tables

toolkit, which is followed by the measurements in Section 4. Finally, in Section 5 we sum up the main findings and describe directions of future work.

2 Combinatorial ranking and unranking

In this Section, we discuss several combinatorial ranking/unranking implementation directions. In particular, we shall look into the following techniques:

- Precomputed lookup tables.
- Combinatorial unranking over lexicographically ordered bitmaps (lex).
- The above over co-lexicographic order (colex).

Other combinatorial unranking algorithms, like Cool-lex[4], are for further study.

2.1 Lookup tables

The trivial method of combinatorial unranking is to use two lookup tables (see Fig. 2): an OFFSET_TABLE that stores class offsets, and a BITVECT_TABLE that stores the bit vectors themselves. The OFFSET_TABLE holds exactly b (block size) plus one elements, as there are $b + 1$ classes[1]. The BITVECT_TABLE has $\binom{b}{0} + \binom{b}{1} + ... + \binom{b}{b-1} + \binom{b}{b} = 2^b$ number of elements because it stores all combinations of bit vectors. The pseudo-code for trivial implementation is given in Fig. 3a.

As we increase the block size b, the exponential growth of memory consumed by BITVECT_TABLE limits its usage. On smaller scales, when the full table fits into the processor cache[2], lookup tables can be very fast, while on larger scales the lookup table itself might not entirely fit into the main memory, causing cycle hungry memory swapping. Easily, there is a middle ground where only a portion of the table fits into the processor cache but it otherwise fits into the RAM entirely. We shall take a closer look on the performance of lookup tables in this particularly important and interesting operational regime in Section 4.

[1] Plus one, because of the zero class which means none of the bits is set.
[2] In this paper, we mean last level cache if it is not otherwise stated.

```
uint_fast8_t
table_unrank_onebit(
  uint_fast64_t rank,
  uint_fast8_t weight,
  uint_fast8_t bit) {
  return !!(
    bitvect_table[offset_table[weight]+rank]
    & (uint_fast64_t)1<<bit);
}
```

(a) Bit vector lookup table bit unrank (table-bit-onebit)

```
uint_fast8_t
lexico_unrank_onebit(
  uint_fast64_t rank,
  uint_fast8_t weight,
  uint_fast8_t width,
  uint_fast8_t bit) {
  uint_fast64_t v;
  while(weight) {
    // shifted binom table
    v = binom[width][weight];
    if(v>rank) {
      if(!bit) return 1;
      --weight;
    } else rank -= v;
    if(!bit--) return 0;
    --width;
  }
  return 0;
}
```

(b) Optimized lexicographic bit unrank (lex-bit-opt3-onebit)

Fig. 3: Bit unrank implementations for table lookup and lex-oder.

2.2 Lex

Interestingly, the mapping from class-offset pairs to bit vectors (and vice versa) can be done algorithmically. The idea is as follows. Suppose we need to enumerate all b-bit bitmaps of class c (recall, this means that there are c bits set to 1 in the block) and determine the position of any given bitmap t in this enumeration (i.e., we do combinatorial ranking). Instead of the bitmap t, we are going to rank its position vector $[t_1, t_2, \ldots, t_c]$, which encodes the position of 1s within t. Easily, there are $\binom{b}{c}$ bitmaps on class c and the same number of position-vectors, so the mapping from bitmaps to position vectors and reverse is one-to-one.

Our enumeration (for now) will be based on the ordering of position vectors of class c in lexicographic order. Now, given a position vector $[t_1, t_2, \ldots, t_c]$, we need to determine how many position vectors there are in the lexicographic order *before* this vector. This is the sum of all the position vectors $[u_1, u_2, \ldots, u_c]$, with $u_1 < t_1$, or $u_1 = t_1$ but $u_2 < t_2$, or $u_1 u_2 = t_1 t_2$ but $u_3 < t_3$, etc. One easily sees that the number of such position vectors (and so the rank r of t) is

$$r = \sum_{i=1}^{c} \left(\sum_{j=t_{i-1}+1}^{t_i - 1} \binom{b-j}{c-i} \right) .$$

Unranking, that is, determining the position-vector given its rank r in the lexicographic order, is a bit more involving. The main idea is that we go through all $i = 1, \ldots, c$ and all particular positions $j = 1, \ldots, b$ and we decide whether or not the i-th bit set to 1 can appear in position j. This can only happen if $r > \binom{b-j}{c-i}$. For a more in-depth discussion, refer to [3, 5, 6].

```
uint_fast64_t
colex_unrank_bitvect(uint_fast64_t rank, uint_fast8_t weight) {
  int_fast8_t i=weight-1, p;
  uint_fast64_t res=0, mask=1;
  while(i>=0) {
    p=i;
    while(binom[p][i+1]<=rank)
      ++p;
    res |= mask<<(p-1);
    rank-=binom[p-1][i+1];
    i-=1;
  }
  return res;
}
```

Fig. 4: Colexicographic bit vector unrank (colex-bit)

2.3 Colex

Colexicographic unranking is similar, but it is based on a colexicographic order instead of a lexicographic order. Here, colex order is essentially a reversal of the lex order. Interestingly, just this simple modification creates a whole lot of optimization chances, as we shall see in the next Section when we discuss our implementations.

3 Implementations and optimization

We present optimized lookup table based, lex and colex unranking algorithms. Optimization is performed in an iterative way, trying out multiple tactics and verifying the results by a profiler. We aim to use as few variables, and operations on them, as possible, and eliminate temporary values by using increment and fetch operator. We also take care of counting towards zero in loops so we cut down cycles. Our working platform has an x86-64 instruction set, so we do not have to address penalty caused by function parameter passing. The compiler always passes parameters in registers, which would conventionally require stack operations (because of the small register file).

3.1 Table

For the lookup table we only present the bit access code (see Fig. 3a), because getting the bit vector is trivial. Double logical negation is used on the masked bit vector for getting proper boolean output (zero or one) instead of zero or many. One negation can be saved by storing a table of inverted bit vectors, however in this case compiler's optimization fails and generates a longer assembly code. This code can be also written without logical negations:
`bitvect_table[offset_table[weight]+rank]>>bit & (uint_fast64_t)1` .
This is what the compiler does anyway after optimization.

```
uint_fast64_t
lexico_unrank_bitvect(
  uint_fast64_t rank,
  uint_fast8_t weight,
  uint_fast8_t width) {
  uint_fast8_t i=0, t=0;
  uint_fast64_t v, res=0, mask=1;
  while(i<weight) {
    v = binom[width-1-t][weight-i-1];
    if(v>rank) {
      res |= mask<<t;
      ++i;
    }
    else rank -= v;
    ++t;
  }
  return res;
}
```

(a) Lexicographic bit vector unrank (lex-bit-normal)

```
uint_fast64_t
lexico_unrank_bitvect(
  uint_fast64_t rank,
  uint_fast8_t weight,
  uint_fast8_t width) {
  uint_fast64_t v, res=0, mask=1;
  while(weight) {
    // shifted binom table
    v = binom[width][weight];
    if(v>rank) {
      res |= mask;
      --weight;
    }
    else rank -= v;
    --width;
    mask<<=1;
  }
  return res;
}
```

(b) Optimized Lexicographic bit vector unrank (lex-bit-opt3)

Fig. 5: Lexicographic bit vector unrank implementations

3.2 Lex

For the lexicographic unranking code, we do the optimization for both kinds of lookups (i.e., the lookup for the complete block and the simple one-bit access form). See Fig. 5. Note that we implement the reverse variant of lex which means it should be read from the less significant bit to the most significant bit (to be in lexicographic order). In the optimized algorithm, the binomial coefficients table has to be shifted bottom-right by one element to be able to save some subtractions.

The function parameters are the following:

– *rank*: the rank value need to be unranked (same as relative offset o for each class c in OFFSET_TABLE lookup table);
– *weight*: the number of ones in the bit vector (previously referred to as the class c);
– *width*: length of the bit vector (i.e., the block size b).

To find whether a bit is zero we iterate until we have found *weight* number of ones. The maximum number of iterations is *width* and each iteration decides one bit of the bit vector. In each iteration we compare a chosen binomial coefficient to the current rank value. If rank is less it means a bit set to 1 in the bit vector, otherwise it is set to zero and rank has to be decreased by this coefficient.

The row of chosen binomial coefficient is started at $width-1$ and decreases in each iteration (may reach row zero if the most significant bit is one). The column of the chosen binomial coefficient is initiated to $weight-1$ and decreases only on bits that are set to 1 (reaching always column zero). This essentially means that the algorithm can walk through $width * weight$ sized rectangle in the binomial coefficients table (see more on this "geometrical" view in [6]).

If we are interested in only one bit, as in Fig. 3b, we can save time by stopping the algorithm in the right column.

In addition to our optimized implementation of *lex-bit-opt3*, we also created an even faster implementation we shall call *lex-bit-opt4*. To save space, we do not present this algorithm separately. The idea is similar, but the code contains optimization for the case when *weight* is larger than the half of *width*, using the fact that $\binom{a}{a-b} = \binom{a}{b}$. This part of the code creates normal lexicographic ordering, so this unrank technique is a mixed variant of reverse and normal lexicographic ordering.

3.3 Colex

The algorithm, presented in Fig. 4, is similar to the lexicographic unrank algorithm, except that it finds the 1s corresponding row by increasing from row zero (instead of the decreasing property of lex) and by this design the *width* parameter is not required. We look for the highest value in the current column of the binomial coefficients table which can be subtracted from the current rank and results in a natural number. Then we step to the previous column and do the same until reaching the last column. Each row iteration during finding that value results in a zero bit, while each column change corresponds to a one bit in the resulting bit vector.

4 Performance evaluation and profiling

In order to measure the performance of our implementations, we created a comprehensive benchmarking framework (called `Cyclecount`) and profiling scripts using the `Callgrind` tool. The `Cyclecount` framework can provide cycle accurate measurements and also creates diagrams for them. `Callgrind` is an extension to `Cachegrind` and part of the sophisticated `Valgrind` profiling tool suite.

`Cyclecount` is based on the excellent measurement library of Agner Fog and it includes a kernel driver and a user space program. The library contains instruction latency and throughput tables [7] for different processor architectures. The elapsed cycles measured using the processor's Performance Monitor Counter (PMC) and the emptiness of the pipeline is guaranteed by using the CPUID [3] instruction. PMC is an internal hardware counter for measuring events like elapsed cycles, cache misses, failed branch predictions, etc. The framework runs the test cases multiple times and calculates mean values to eliminate spikes that may be caused by interrupts. Because we measure clock cycles instead of elapsed time, the frequency scaling property of the CPU cannot alter the results.

For compiling and measuring the code with different parameters we have introduced a special `interval` precompiler macro. This macro is processed by our framework and, besides replacing it with the corresponding precompiler macro

[3] Normal usage of this instruction is to determine features of the CPU, however it also has a property of flushing the pipeline.

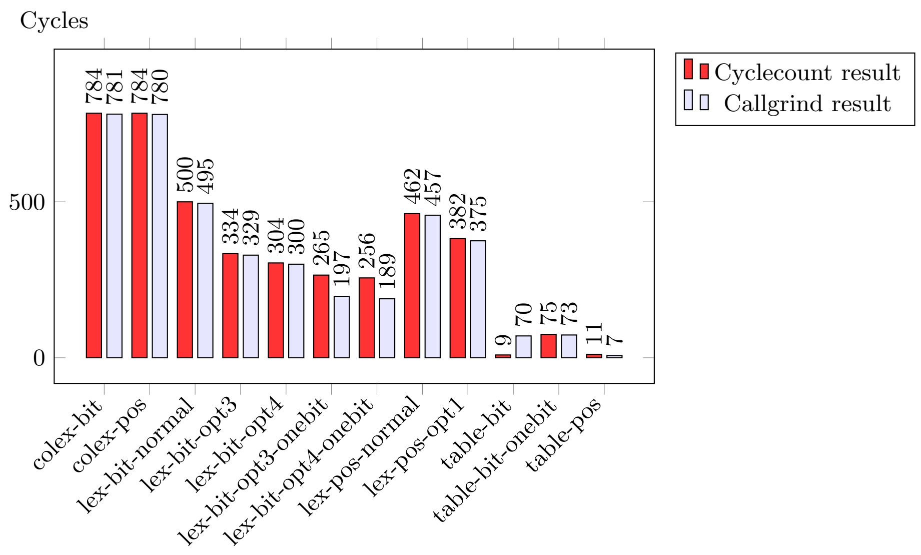


Fig. 6: Performance of unranking techniques (block size=25, random classes)

for optimization, it also automatically orders the tests into groups in the diagram. Multiple output formats are supported, like simple text file for human reading, `XML` file for machine processing, and plot files for diagrams using `GNUPlot` in PNG and EPS formats. `Cyclecount` is written in C++ and Python, and runs on Linux exclusively.

The profiling script uses `Callgrind` to measure the consumed cycles. Unfortunately, `Callgrind` simply returns event numbers but it does not calculate total cycles for us. We used the metric of `KCachegrind` to calculate the estimated cycles caused by cache misses: 10 cycles for a `level-1` miss and 100 cycles for a last-level miss. The profiling script has an option to compare the output of the profiled programs to ensure that the optimized program generates the same output as the normal program. In this case, the seed of the randomization is not the time but a predefined value (12345). The script comes in the form of GNU `Makefiles` and use only standard Unix utilities, hence it should run on any Unix-llike platform where `Valgrind` is available.

The measurements are done on an Intel Core 2 Quad Q6600 processor with Fedora Linux 19 (x64) and GCC 4.8.2 compiler (with `-O2` and `-march=core2` optimizations). The tests are done using random (but valid) ranks and using 1 million unrank operations.

Our benchmarking suite together with all the source codes is available for download at `https://github.com/andmaj/eunice2014article`, so one can independently verify the results.

4.1 Results

In Fig. 6, we present the performance of unranking techniques with random classes but fixed block size using the two performance evaluation tools. The block size b is 25 bytes because in this case the 268MB lookup table would certainly not fit the processor's cache but can be easily fit into the RAM. The values are quite close in most cases so `Callgrind` does a good job on simulating the processor. However in case of *lex-bit-opt3-onebit* and *lex-bit-opt4-onebit* `Callgrind` shows a value 26% lower than the result of the real world `Cyclecount` test and there is a big difference between the *table-bit* results too. We have also implemented the unranking techniques in terms of position vectors[4]. Unfortunately, the results could not be included in the paper due to space limits, so we only sketch some results in Fig. 6.

Fig. 7a shows the dependence on the block size of the unranking performance with random classes, while in Fig. 7b the focus is on class dependence with fixed block size.

4.2 Table

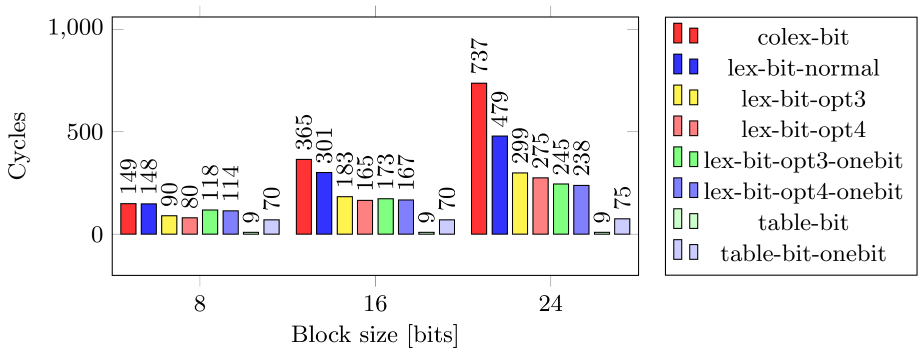
Interestingly, we found the *table-pos* technique to be the fastest amongst the unranking techniques. As it just returns a reference of a position vector, the simplicity of the *table-pos* buys performance. Note that in common applications, where one used to make a copy, it would be slower than other lookup table methods. In other cases, however, table-based unranking is still a very reasonable choice, despite the promise of the other purely algorithmic techniques.

Curiously, we found that a *table-bit* unrank query terminates in only 9 cycles in the `Cyclecount` test, but when bit manipulation takes place on this data (like in the case of *table-bit-onebit*) this jumps to 75 cycles. We believe that there may be an optimization in this processor family on simple memory copy operations, or it can arrange the micro-operations in a way that the query runs quasi-parallel with the random number generation functions. The operations are certainly completed because we have also tested summarizing the returned values. It also seems that the performance of lookup tables is independent of the class and block sizes.

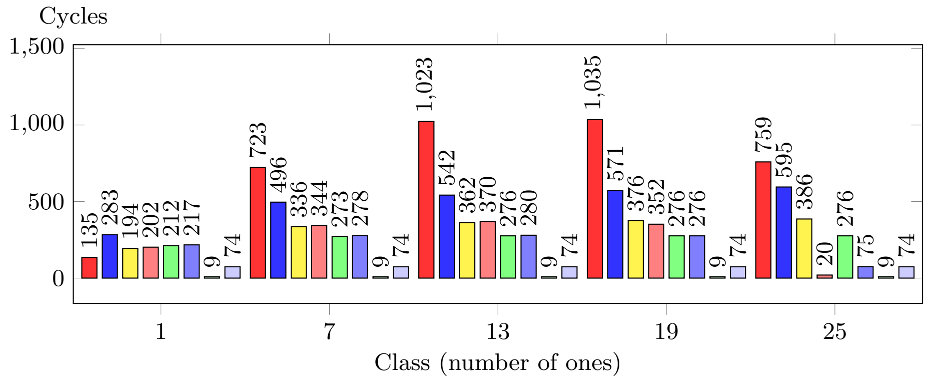
4.3 Lex

Our optimized lex unranking algorithm *lex-bit-opt4* can attain 304 cycles which is 39% faster than the plain version *lex-bit-normal* as seen in Fig. 6. Further, if we are interested in only one bit of the bit vector performs even better, reaching 256 cycles as *lex-bit-opt4-onebit*. The performance of lex shows the same trends

[4] Recall that the difference between a bit vector and a position vector is that a bit vector stands for the bitmap itself and can be represented by a 64 bit integer, while the position vector is an array of 8 bit integers, each representing the position of a bit set to 1.



(a) Different block sizes



(b) Different classes, block size=25

Fig. 7: Performance of unranking techniques

as that of colex for growth of block size, however it increases logarithmically with the class size. However, the optimizations in *lex-bit-opt4* and *lex-bit-opt4-onebit* result in a linear decrease of cycle count with the class parameter, after the class becomes larger than half of the block size.

4.4 Colex

As can be seen in Fig. 6, colex can be considered a very bad choice for combinatorial unranking in terms of speed, with only 784 cycles per unrank operation. The number of CPU cycles consumed by colex grow linearly with block size (see Figure 7a). It is interesting that for different classes it shows a wave like behavior but never forms a half period (see Figure 7b).

5 Conclusion

At the moment, it seems that combinatorial unranking algorithms cannot replace simple lookup tables in compression/de-compression codes, at least as long as the block size is small enough for the lookup table to fit entirely into the RAM. However, they may still find their use for larger block sizes, that is, when the exponential growth of lookup tables would cause the table to overflow the main memory and swapping would ruin the process. For such cases, our optimized combinatorial unranking codes seem to present a decent choice for implementors.

Even so, we have not given up yet the possibility that a hardware-assisted implementation would be able to reach the performance of lookup tables someday. This presents a great opportunity for future research.

References

[1] R. Raman, V. Raman and S. Srinivasa Rao: Succinct Indexable Dictionaries with Applications to Encoding K-ary Trees, Prefix Sums and Multisets ACM Trans. Algorithm. Volume 3, Issue 4, Article No. 43 (2007)

[2] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán and Z. Heszberger: Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond, Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, 111–122 (2013)

[3] Donald E. Knuth: The Art of Computer Programming: Combinatorial Algorithms, Part 1 (2011)

[4] Frank Ruskey, Aaron Williams: The coolest way to generate combinations, Discrete Mathematics, Volume 309, Issue 17, 5305–5320 (2009)

[5] Zbigniew Kokosiski: Algorithms for Unranking Combinations and Other Related Choice Functions, http://riad.pk.edu.pl/∼zk/pubs/95-1-006.pdf (1995)

[6] Ratko V. Tomic: Quantized Indexing: Background Information, `http://www.1stworks.com/ref/TR/tr05-0625a.pdf(2005)`

[7] Agner Fog: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, http://www.agner.org/optimize/instruction_tables.pdf, (2014)

[8] Veli Mäkinen and Gonzalo Navarro: Dynamic Entropy-compressed Sequences and Full-text Indexes, ACM Trans. Algorithms, Volume 4, Issue 3, Article No. 32 (2008)