A heuristic approach to find short efficient WOM codes

Bert Dobbelaere
(bert.o.dobbelaere at telenet dot be)

Abstract

In this document, some methods are presented that use a heuristic scoring function and incremental improvement cycles as means to obtain short efficient Write-Once Memory (WOM) codes. While initially only fixed rate two-write WOM codes were targeted, the approach has also successfully been applied to unrestricted rate cases, and additionally to 3- to 6-write codes. As extension, the principle of incremental improvement has also been applied to the "*coset coding*" method starting from a random parity check matrix. Several practical results were obtained, including two-write codes $<28>^2/7$, $<127>^2/10$, $<2144>^2/16$ and $<176,94>/10$, three-write codes $<7,12,8>/6$ and $<2^6>^3/11$ (fixed rate 1.636), and some convenient fixed rate four-write codes ranging from $<2^4>^4/10$ to $<2^8>^4/18$, the latter providing a fixed coding rate of 1.778. Also $<2^6>^5/16$ and $<2^5>^6/16$ have been found, both with a fixed coding rate of 1.875. Using coset coding and allowing a "known write number", fixed rate results match or improve those from [YKS+10], including codes of sizes $<2^7>^2/10$, $<2^{13}>^2/18$ and $<2^{16}>^2/22$. The highest unrestricted rate obtained in this way for the two-write case was 1.50736 for a 40 bit code. In the final chapter, an excursion is made on how to improve non-guaranteed write performance.

1 Introduction

In their pioneering work of 1982 [RS82], Rivest and Shamir introduced the concept of a WOM code and related terminology, its theoretical base, and provided next to the renown $(<4>^2/3)$-WOM code a different 2-write example, which was a computer generated solution for a $<26>^2/7$ code. The authors could also prove the non-existence of a $<29>^2/7$ code, but were unable to find a $<27>^2/7$ code. The question whether a $<27>^2/7$ or even a $<28>^2/7$ code existed remained unsolved. Triggered by the open question on the "*Ponder This*" pages hosted and maintained by IBM research (March 2015 challenge), I decided to try to find a solution for a $<27>^2/7$ code. Initially failing to reproduce even the $<26>^2/7$ case, it was directly obvious that any exhaustive search inspired attempt to solve the $<27>^2/7$ case was very unlikely to succeed. Introducing an incremental optimization approach based on a heuristic gave much better results, including an unexpectedly fast solution for the $<27>^2/7$ case. Encouraged by this result, I tried the same approach on the $<28>^2/7$ problem. Improving the algorithm again with a mechanism to prevent premature lock-up of the incremental improvement cycle, a solution was also found for the latter case, solving the open problem from 1982. Assuming as 28 symbols the letters A…Z, a dot and a space, the "seven-track paper tape" example from [RS82] now could look like Table 1. The decimal representation of the binary encoded pattern is the sum of the row and column labels. The eligible "first write" positions correspond to 28 of the possible 29 patterns having a

Hamming weight ≤ 2 and are marked with bold/underline. The rate of this code is $(2 \cdot \log_2(28)/7) \approx 1.3735$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	**A**	**B**	**C**	**D**	**E**	**F**	**G**	E	**H**	**I**	**J**	R	**K**	V	D	X
16	**L**	**M**	**N**	Q	**O**	I	V	space	**P**	E	V	Z	U	W	Y	S
32	**Q**	**R**	**S**	V	**T**	Q	F	Y	**U**	T	E	F	O	M	Z	N
48	**V**	P	D	X	E	A	H	J	Q	K	space	G	B	L	R	.
64	**W**	**X**	W	M	**Y**	.	L	O	**Z**	Q	K	U	E	J	Q	B
80	space	V	E	L	Q	R	P	K	.	N	M	A	G	H	F	T
96	.	E	Q	I	S	G	K	W	R	S	L	H	V	space	A	P
112	S	F	T	B	N	Z	M	U	J	Y	W	O	X	D	I	C

Table 1: Symbol coding table for a <28>²/7 code

The approach used to obtain this result was first tried on fixed rate 2-write WOM codes of different length, and because the subject of WOM codes goes far beyond the aforementioned subclass, it was tempting to try to extend the method to a few other categories. Sometimes solving a problem raises far more questions than it answers.

All code examples mentioned in this document have been found using the heuristic techniques described. If a code of identical dimensions was reported before, this simply means that its performance can be matched using the described methods.

2 Basic method for the unrestricted binary two-write case

Although initially applied to fixed-rate codes, the method is presented here for the unrestricted case, as only a few optimizations depend on the fixed nature of the code. The algorithm is in essence a *hill climber* based on random mutations.

The basic algorithm

In what follows we assume a code of length n, and $\{v_1 \ldots v_t\}$ the number of different symbols that we want to be able to write during the t writes respectively (t=2 in this chapter). v_{max} is the maximum value of $\{v_1 \ldots v_t\}$. The space of encoded patterns $\{0,1\}^n$ and the integer range $\{0 \ldots 2^n\text{-}1\}$ will be used interchangeably based on binary representation. We assume that each pattern $p \in \{0 \ldots 2^n\text{-}1\}$ maps to a symbol $s_p \in \{0 \ldots v_{max}\text{-}1\}$, and the symbols that are written during write i are integers in the range $\{0 \ldots v_i\text{-}1\}$. The algorithm starts with a mapping from each pattern to the symbol 0. In each iteration, a random mutation will be applied to the current mapping. Two types of mutation are defined, each applied with 50% probability. The first mutation type (“point mutation”) replaces the mapping for a random pattern with a random symbol. The second mutation type (“swap”) exchanges the mappings of two random patterns. The mutated mapping is accepted as base for the next iterations if it improves or at least equals the best value of a heuristic score function so far. The score function (see later) is designed to return its maximum value iff a solution is found.

Algorithm Find_Basic_Solution
output: pattern to symbol mapping solution

$s_p \leftarrow 0$, for p in range [0…2^n-1]
bestscore ← 0
while bestscore < maxscore(v_1,v_2), **do**
$r_p \leftarrow s_p$, for p in range [0…2^n-1]
mut_type ← random(2)
if mut_type = 0 **then**
p ← random(2^n)
$r_p \leftarrow$ random(v_{max})
else
p ← random(2^n)
q ← random(2^n)
$r_p \leftarrow s_q$
$r_q \leftarrow s_p$
if score(r) ≥ bestscore **then**
$s_p \leftarrow r_p$, for p in range [0…2^n-1]
bestscore ← score(r)
return s

The random(x) function used in the algorithm above returns a random integer in the range {0…x-1}. The performance of the algorithm can be slightly improved at the cost of some additional complexity by rejecting “mutations” without effect (replacing a symbol by its original or swapping two identical symbols)
A modification that in practice gives good results is to replace the “swap random positions” mutation by a “swap random neighbours” mutation, “neighbours” being points with Hamming distance 1. In this case, the statement “q ← random(2^n)” is replaced by “q ← p ^ $2^{random(n)}$”, with “^” the bitwise exclusive or operator. Although less alterations can generated by only swapping neighbours, the probability for a random neighbour swap to be accepted as “improvement” can be much higher than the the probability that a swap of two random positions is accepted.

The score function

The score function evaluating a mapping needs to be an (at least rough) indicator of how close its argument is to a solution. Based on the feedback it provides, the evolving sequence of future inputs should be directed towards higher scores, and yet its execution time should remain limited as it will be called during each iteration. The score function used here is a heuristic based on the number of (primary symbol, secondary symbol) pairs supported by the mapping that is provided as its argument. The minimum score that can be returned is 1 (corresponding to the initial mapping), the maximum is the product of the number of symbols to be supported during each write phase, so maxscore(v1,v2) = $v_1.v_2$. If maxscore(v1,v2) is reached, we found a <v_1,v_2>/n WOM code. The score function takes into account that the only valid patterns corresponding to a first write must have a Hamming weight low enough to allow v_2 different patterns during the 2^{nd} write, hence the maximum

Hamming weight HW_{max} for a first write symbol is given by $HW_{max} = \lfloor n - \log_2(v_2) \rfloor$

Algorithm score_basic

input: pattern to symbol mapping
s={s_p}, p in [0..2^n-1]
output: score value

$a_i \leftarrow 0$ for i in range [0..v_1-1]
for p in range [0.. 2^n-1] **do**
if HW(p)≤ HW_{max} **then**
$b_i \leftarrow 0$ for i in range [0..v_2-1]
for q in range [0.. 2^n-1] **do**
if (q&p) = p **then**
$b_{s_q} \leftarrow 1$
sum ← $\sum_{i=0}^{v_2-1} b_i$
if sum ≥ a_{s_p} **then**
$a_{s_p} \leftarrow$ sum
sum ← $\sum_{i=0}^{v_1-1} a_i$
return sum

HW(x) denotes the Hamming weight of the binary representation of x. The operator "&" performs the bitwise ***and*** function. Hence the expression "(q&p)=p" simply means that q contains a superset of the bits that are set in p, meaning the pattern q can be written after p was written first. The basic score function presented above is computationally rather expensive as it iterates over the pattern range in two nested loops. Optimized variants can be made that are much faster. The variant below back-propagates the covered set to adjacent patterns with a lower Hamming weight:

Algorithm score_fast1

input: pattern to symbol mapping
s={s_p}, p in [0..2^n-1]
output: score value

$descendants_p \leftarrow 0$ for p in range [0..2^n-1]
for p in 2^n-1 **downto** 0 **do**
k ← s_p
mask ← 2^k | $descendants_p$
for b in [0..n-1] **do**
idx ← p & (2^n-1-2^b)
$descendants_{idx}$ ← $descendants_{idx}$ | mask
$maxval_i \leftarrow 0$ for i in range [0..v_1-1]
for i in [0… #(first_write_patterns)-1] **do**
idx ← $first\_write\_patterns_i$
val ← HW($descendants_{idx}$)

```
            k ← s_idx
            if val > maxval_k then
                maxval_k ← val
sum ← Σ_{k=0}^{v_1−1} maxval_k
return sum
```

The operators “&” and “|” perform the bitwise ***and*** and ***or*** functions. The *descendants* array elements and the *mask* variable need to hold at least v_{max} bits. “*low_hw_patterns*” is an array of constants containing the patterns with Hamming weight ≤ HW_{max}. The inner “for” loop can be unrolled and its constant expressions can be precalculated. A considerable fraction of the assignments in the inner loop could be avoided in cases where idx=p, but at the cost of additional control flow complexity[1].

A still faster variant of the score function for larger n values is implemented in the example program WOM_finder. That program exploits the fact that the mutation (“point mutation” or “swap”) leaves most code points unaltered, hence the coverage of supported (1st write, 2nd write) symbol pairs by the code can be updated in an incremental way each iteration.

3 Mitigation of search lock-up

The basic algorithm provided in the previous chapter works well for many cases. E.g. the $<27>^2/7$ code, undiscovered since 1982, was found by a simple unoptimized C implementation in far less than a second executing on a single CPU core. However, there are some serious restrictions too. The aforementioned score function is mapping a 2^n dimensional space of integers in range $[0…v_{max}-1]$ to a single integer, the function being able to detect a perfect score, but in some cases even changing a single element of the input can have a large effect on the output. It is to be expected that, when trying to find a global maximum of the score function in this way, there is no guarantee that such maximum can be reached by going only “uphill”. The first crucial mitigation was already part of the basic algorithm: mutations don’t have to bring an improvement in order to be accepted. It is sufficient that they match the best score so far. This improves the “mobility” in the 2^n-dimensional search space, offering better chances to eventually find an upward path later. Even so, in many cases this lets the search algorithm get stuck in an subspace from which there is no escape without going down. There are several techniques to improve the chances of finding a global maximum, e.g. simulated annealing, tabu search,… The method I initially applied to find a $<28>^2/7$ solution used a repeated restart at a random position, with filters that weed out searches that fail to reach a certain score level after a certain amount of iterations. The filters were adapted to the population in the sense that a fixed fraction of the searches were allowed to continue at different stages. With this technique, it still took about 150 thread-hours on a Intel® Core i7-4770K to find a solution for the $<28>^2/7$ case. Better

1 On modern processor architectures, computing a branch and conditionally execute a mere few instructions is usually slower than executing those instructions always, so the latter is preferred if those instructions don’t have unwanted side-effects.

results were obtained in the WOM_finder program by making the score function itself adaptive.

The score_basic algorithm contains at the end the statements

$$\text{sum} \leftarrow \sum_{i=0}^{v_1-1} a_i \quad \text{; return sum.}$$

This makes that the score returned is the sum of the maximum number of 2nd write symbols found for each 1st write symbol. In the case of a solution, that maximum (a_i) equals v_2 for all i, the resulting return value being $v_1 \cdot v_2$. The score function was updated by introducing weights for each 1st write symbol, the above statements becoming

$$\text{sum} \leftarrow \sum_{i=0}^{v_1-1} a_i \cdot w_i \quad \text{; return sum.}$$

The integer weights are selected so that $w_i \geq 1$ for all i , and

$$\sum_{i=0}^{v_1-1} w_i = v_1 + bonus$$

In the case of a solution, the return value now becomes $(v_1 + \text{bonus}) \cdot v_2$ and the definition of the `maxscore` function is updated accordingly. The “bonus” is randomly distributed over the 1st write symbols.

Since the weights are all positive, a solution is still located ‘at the top of the hill’. However, the priorities can be altered: if e.g. a certain mutation caused an improvement of 2 in the number of reachable 2nd write symbols for a given 1st write symbol, and a deterioration -3 for a different 1st write symbol, this mutation would originally – ceteris paribus – be rejected, while with the introduction of the weight function it is now acceptable in some cases. The idea behind the weight function is that the weights can periodically be reassigned (e.g. each 10000 mutations, random weight values), so that changing priorities let the algorithm explore all sub-targets. The mutations that are of benefit for a large fraction of the score functions[2] are now more likely to prevail.

The method can to some extent be compared to compacting loose sand grains in a bucket. Trying to push the sand down won’t do much good, but softly shaking the bucket in different horizontal directions will certainly help.

A solution for the $<28>^2/7$ case now takes around 24 thread-hours, still uncomfortably long, but a significant improvement over the original. Some highlights of the various codes that have been found using this technique have been listed in Table 2

2 The weights here considered as defining a set of score functions, not as additional arguments

Code length (n)	1st write symbols	2nd write symbols	Fixed rate	Sum rate
7	28	28	1.3735	1.3735
9	130	47	1.2344	1.3974
10	127	127	1.3977	1.3977
10	176	94	1.3109	1.4014
11	195	195	1.3832	1.3832
15	1280	1280	1.3763	1.3763
15	4944	512	1.2	1.4181
16	2144	2144	1.3833	1.3833

Table 2: 2-write WOM results using arbitrary decoding map

A more extensive list of results, together with matching decoding tables and the "*WOM_finder*" program for generating them, is available on [WOM page].

4 Extension to multiple write codes

The definition of the score function in the case of t writes can be easily extended: it now counts the number of (s_1, s_2, … s_t) symbol combinations that are supported by the code over the t writes (in its basic form). The "bonus" from previous chapter can also be introduced in this case. Good results were achieved for 3-write and 4-write codes by making the bonus only dependent on the 1st write symbol, so the score function is still computed as $\sum_{i=0}^{v_1-1} a_i \cdot w_i$, with this time a_i defined as the number of (i, s_2, … s_t) symbol combinations supported by the code. The maximum score then becomes $(v_1 + bonus) \prod_{r=2\ldots t} v_r$.

For computing the number of "supported" symbol combinations, for each of the t writes only those patterns are considered that are eligible to be written in the corresponding write as part of an actual solution. The list of candidate patterns can be precomputed and obtained as follows:

- All 2^n patterns are eligible for the last write t
- The patterns eligible for write t-1 are those for which at least v_t last write patterns are possible
- The patterns eligible for write t-2 are those for which at least v_{t-1} patterns eligible for write t-1 are possible.
- …
- The patterns eligible for write 1 are those for which at least v_2 patterns eligible for write 2 are possible.

Obviously a solution is only possible if the number of eligible patterns for the 1st write is ≥ v1. It is easy to see that each of the pattern sets eligible for a given write number corresponds to the patterns bound by a maximum Hamming weight, which increases with the write number. Some codes obtained using this technique have been listed in Table 3.

Number of writes (t)	Code length [bits] (n)	Symbols written	Fixed rate	Sum rate
3	5	6,5,6	1.3932	1.4984
3	6	7,12,8	1.4037	1.5654
3	8	20,15,15	1.4651	1.5170
3	9	24,24,24	1.5283	1.5283
3	10	36,36,36	1.5510	1.5510
3	11	(*) 56,56,56	1.5838	1.5838
4	9	10,14,14,14	1.4764	1.6382
4	10	16,16,16,16	1.6	1.6
4	12	32,32,32,32	1.6667	1.6667
4	14	64,64,64,64	1.7143	1.7143

Table 3: Multi write codes obtained by extension of the score function

All codes listed above can be decoded without knowing the write number. (*) The result for 3 writes in 11 bit will be further improved using the hybrid technique described in chapter 6.

5 Application of heuristic search to coset-coding

The concept of coset-coding was introduced by Cohen et al. in [CGM86]. Efficient two-write WOM codes based on a random parity check matrix were reported by Yaakobi et al. in [YKS+10]. The highest reported fixed rate found by computer search using their approach was 1.4546, the highest unrestricted rate 1.4928. A fixed rate of 1.375 was also reported for a constructed 16 bit code. Codes reported in [YKS+10] assume a “known write number”, which has a negligible effect on the coding rate when considering a large number of blocks being written together. The method involves in essence counting the number of full-rank submatrices that are contained in the parity check matrix, a time-consuming task for larger codes. The number of possible submatrices (obtained by omitting columns) is $\sum_{j=0\ldots k} \binom{n}{j}$ for a code of length n and supporting 2^{n-k} second write symbols.

The number of full-rank submatrices found is the number of 1st write symbols supported by the code (assuming ‘known write number’) and hence direct metric of how well a given parity check matrix performs when trying to optimize the number of first write symbols.

A variant with unknown write number computes the syndromes of the potential 1st write patterns found as described, and then reduces this set allowing each syndrome to occur only once. In this case the metric becomes the number of different syndromes usable as 1st write symbol.

The most obvious algorithm to achieve a well-performing matrix would be a simple trial and error, evaluating a large number of random matrices and retaining the “best” one. The method I used to search codes is based on the following observations:

1) There is at least a weak relation between the performance of a parity check matrix and a modified matrix that is altered in one of the following ways:
   - modification of a low number of elements
   - applying a column addition
   - applying a row addition (‘unknown write number’ case)

   For the 1st and 2nd alteration types, the submatrices containing only unaffected columns will yield the same result as before, while the 3rd type can influence the number of collisions in the ‘unknown write number’ case, but will not affect the number of full rank submatrices.
2) It is possible to perform an estimate which candidates of a group of parity check matrices contain the highest number of full rank submatrices. To achieve this, a statistical sample of submatrices can be evaluated much faster than is required for a full count. Multiple ‘selection rounds’ using increasing sample sizes can be set up in order to improve the average score of the population.

In order to find a “good” parity check matrix, we start with chosing a random matrix, and its number of 1st write symbols supported is computed by counting the full rank submatrices (or in case an unknown write number is desired, the related number of non-colliding syndromes).

A large number of mutated matrices are then produced, and preselection rounds based on statistical sampling of submatrices are performed, improving the average fitness of the remaining candidates. For a matrix coming through all preselections, again a full count is performed. If it improves or matches the initial matrix, it is used as “parent” for a new set of mutations etc.

Some results obtained for the “unknown write number” and “known write number” cases are provided in Table 4 and Table 5 .

Code length (n)	1st write symbols	2nd write symbols	Fixed rate	Sum rate
18	6386	2^{13}	1.4045	1.4245
22	52530	2^{16}	1.4255	1.4400
25	244730	2^{18}	1.4321	1.4360
30	3214262	2^{22}	1.4411	1.4539
36	58106045	2^{27}	1.4329	1.4664
37	121762880	2^{27}	1.4519	1.4557

Table 4: "Coset coding" results for 2-write codes, unknown write number

Code length (n)	1st write symbols	2nd write symbols	Fixed rate	Sum rate
10	(**) 130	2^{7}	1.4	1.4022
10	304	2^{6}	1.2	1.4248
14	(**) 1058	2^{10}	1.4286	1.4319
16	(*), (**) 5065	2^{11}	1.375	1.4566
18	(**) 8380	2^{13}	1.4444	1.4463
22	(**) 67522	2^{16}	1.4545	1.4565
26	(**) 564962	2^{19}	1.4615	1.4657
30	(**)4722654	2^{22}	1.4667	1.4724
34	(**) 39012018	2^{25}	1.4706	1.4770
34	1135925814	2^{21}	1.2353	1.5024
38	(**) 341410434	2^{28}	1.4738	1.4828
40	10533443416	2^{27}	1.35	1.5074

Table 5: "Coset coding" results for 2-write codes, known write number

(*) A code of these exact dimensions has been reported in [YKS+10]. This shows that the heuristic method can – at least in this case – match a result found by construction.

(**) Fixed rate code can be obtained by reducing the number of used 1^{st} write symbols

A more extensive list of results, together with matching parity check matrices, is available on [WOM page].

6 Hybrid approach for multiple write codes

The direct use of a score function as sole mechanism to determine all 2^n symbols in the decoding map for a code of length n is only practical for low n. The following hybrid approach for a t-write code is usable in $t \geq 3$ cases in which 2^k symbols are supported for the last write, and has been successfully applied to produce high fixed-rate results on 3- to 7-write WOM codes

- Using the optimization method described in the previous chapter, find a $k \times n$ parity check matrix with a high amount of 1^{st} write vectors for the 2-write case. For each bit vector v for which the Hamming weight $HW(v) \geq n - k$, use the parity check matrix

to compute the syndrome of v and pre-assign it as its symbol.

- Using the heuristic search principle from chapter 4, apply only mutations to the vectors for which HW(v) < n - k .

The latter search is now performed over a restricted range, covering only vectors that are eligible for write numbers smaller than t. The parity check step is far more efficient in finding candidate points for the penultimate write. The restriction that for each such vector, all 2^k reachable symbols in write t need to be different, makes it hard to apply mutations for HW(v) ≥ n - k that don't result in deterioration of the score. Some codes that have been found using this technique are listed in Table 6. All these codes have a single decoding map, i.e. the decoder does not need to know the write number. Several constructions exist for high rate multiple write WOM codes, e.g. the ones in [YKS+12] for unrestricted rate, and [WJ10] for fixed rate. The search here was focused on achieving high fixed rate results. The fixed rates achieved are the highest so far for codes of this length.

Number of writes (t)	Code length [bits] (n)	Symbols written	Fixed rate	Sum rate
3	8	16,16,16	1.5	1.5
3	11	67,68,64	1.6364	1.6503
3	12	72,160,64	1.5	1.6243
4	16	128,128,128,128	1.75	1.75
4	18	256,256,256,256	1.7778	1.7778
5	9	8,8,8,8,8	1.6667	1.6667
5	14	32,32,32,32,32	1.7857	1.7857
5	16	64,64,64,64,64	1.875	1.875
6	10	8,8,8,8,8,8	1.8	1.8
6	16	32,32,32,32,32,32	1.875	1.875
7	15	16,16,16,16,16,16,16	1.8667	1.8667

Table 6: Multiple write results using hybrid approach

7 Squeezing out more

All WOM codes discussed thus far provide a guaranteed number of writes for the given set of symbols. Fixed rate unsynchronized codes have the advantage that, besides the simpler decoding process, it is possible to improve the average number of supported writes for high entropy data by using smart selection at the encoder side: if multiple coding options are available for the next write, we select the one that offers the highest average number of remaining writes (usually one of the options having the lowest possible Hamming weight), bound to the restriction that the number of remaining guaranteed writes is respected. In this way it can be possible to perform one or more "best effort" writes on top of the t writes that are guaranteed by the code.

The problem remains of course, how to detect whether the extra write has succeeded?

Keeping an additional bit per block as indicator severely affects the coding rate for short codes, while clustering the codes and sharing a single indicator reduces the probability that a "best effort" write is possible for all members of the cluster. The obvious solution is to reserve the code pattern for which all n bits are set to '1' as the "dead code" or "failed write" indicator. This requires only a slight modification in the hill climber code: only patterns $\{0\ldots 2^n\text{-}2\}$ are eligible for mutation. If we can produce a fixed-rate code in which we can avoid the all '1' pattern, the option to store "best effort" symbols after the t guaranteed writes comes without dedicated storage bits. It is easy to see that this effectively transforms a fixed rate $<m>^t$ / n code into "$<m>^{t-1}<m+1>$ / n", the "+1" indicating the support for the additional "failed write" symbol as last write[3]. When an amount of data needs to be written to a high number of code blocks, the blocks that cannot be written are set to all '1' and are just skipped by the encoder.

To illustrate the process, assume we use a $<2^7, 2^7, 2^7, 2^7+1>$ / 16 code to store 7 bit data items (for this case a code with "failed write" support was found). We have N 16 bit blocks available, which means that for each of the first 4 writes, we can store N data items, each using up ¼ of the guaranteed fixed coding rate of 1.750. In a 5th "best effort" write, simulation on this code shows that in the case of uniform random data, we can store about 0.634 N data items in the same blocks (assuming sequential access), and still retrieve them using a very simple decoding process (single lookup of the code pattern in a 64K × 7 bit decoding table, skip block if the pattern is 0xFFFF). A 6th write would add another 0.157 N data items. The total coding rate using these 6 writes then becomes about 2.096 (ignoring later writes with marginal contribution).

However, this code still contains many degrees of freedom. An effort was made to improve its "best effort" rate, while preserving its fixed rate property for the guaranteed writes. Again, a heuristic method was used, starting from the code in previous paragraph. Using the same "point mutation" and "swap" random mutations, we work on the code with the "score" function now redefined as follows:

- If the code does not correspond to a valid t write fixed rate code for the m symbols, the score is 0 (a harsh measure to retain the most desirable property)
- If the code corresponds to a valid t write fixed rate code for the m symbols, the score is the statistically expected number of writes NW(s,m,0) supported by the code s

The NW(s,m,p) function is defined as the average number of remaining writes that the code s (an array of 2^n symbols in range [0..m]) supports when starting at the written pattern p. The symbol m is reserved for the last pattern ($p=2^n\text{-}1$) of the code, for all others we have $0 \le s_p < m$.

If a pattern p cannot be altered so it decodes into a different symbol ≠m (most likely its Hamming weight being n-1 already), the symbol s_p is the only symbol that can be written. The probability that this happens the next write is 1/m, and the probability this happens for

3 Unfortunately, in the case of a $<2^k>^t$ / n code, the codes with Hamming weight n-k can not longer be used for the penultimate guaranteed write. Max Hamming weight for penultimate write now becomes n-k-1.

the next two writes is $1/m^2$ etc… Therefore, in this case $NW(s,m,p) = m^{-1} + m^{-2} + m^{-3} + \ldots = 1/(m-1)$

For each write i, there is a probability 1/m that we want to write the symbol k_i. The write will succeed if we find a pattern q that can be obtained by altering p for which $s_q=k_i$. The special case just described corresponds to $k_i = s_p$. For each possible value $k \neq s_p$, we now find the pattern q_k, reachable from p, for which $s_q = k$ and $NW(s,m,q_k)$ is maximized under this constraint. If such q_k is found, then the contribution to the expected number of writes of a transition to symbol k during the next write is $(NW(s,m,q_k)+1)/m$. Taking into account that we may first write s_p again any number of times and then k, the contribution of a transition to k becomes $(NW(s,m,q_k)+1) / (m-1)$. If q_k does not exist, there is no contribution. Adding up all the probabilities, we get

$$NW(s,m,p)=\frac{1}{m-1}\left[1+\sum_{k\in[0..m-1]\setminus\{s_p\}} if\left(\exists q:(q\,\&\,p=p)\wedge(s_q=k)\right)\left(1+max_{q:(q\,\&\,p=p)\wedge(s_q=k)}NW(s,m,q)\right)\text{ else }0\right]$$

with the operator "&" meaning "bitwise and". NW(s,m,0) can be obtained free of recursion by computing NW(s,m,p) for all p decreasing from $2^n - 2$ down to 0. Obviously if s represents a valid $<m>^t$ / n code, $NW(s,m,0) \geq t$.

An optimization run of $12 \cdot 10^6$ cycles was performed on the originally obtained code in an attempt to optimize the "best effort" coding rate. Simulation shows that the number of items written during the 5th write was improved from 0.634 N to 0.692 N, and for the 6th write from 0.157 N to 0.183 N, while maintaining the guaranteed 4 writes for all symbol combinations (see Table 7). If we drop the requirement that the 1st 4 writes must be guaranteed, the rate can be slightly improved again. The performance of a pure random code has been added for comparison. The cumulative coding rates are provided in Table 8. All codes have a "failed write" detection built in.

In an application that relies on the predictable performance of the "best effort" writes, it may be recommended to scramble low entropy data streams (in the case of sequential access) or to apply a transformation that depends on storage location (random access).

Write number	Original $<2^7>^4$ / 16 code	Optimized $<2^7>^4$ / 16 code	Random code	Optimized "best effort" code
1	100.000%	100.000%	100.000%	100.000%
2	100.000%	100.000%	100.000%	100.000%
3	100.000%	100.000%	99.990%	100.000%
4	100.000%	100.000%	89.377%	99.984%
5	63.450%	69.225%	41.746%	71.356%
6	15.731%	18.346%	8.996%	19.246%
7	1.979%	2.406%	1.079%	2.543%
8	0.157%	0.196%	0.085%	0.208%
9	0.009%	0.011%	0.005%	0.012%
10	0.0004%	0.0005%	0.0002%	0.0006%

Table 7: Fraction of successful write attempts (10^8 random trials)

Write number	Original $<2^7>^4$ / 16 code	Optimized $<2^7>^4$ / 16 code	Random code	Optimized “best effort” code
4	1.7500	1.7500	1.7035	1.7499
5	2.0276	2.0529	1.8861	2.0621
6	2.0964	2.1331	1.9255	2.1463
10	2.1058	2.1446	1.9306	2.1584

Table 8: Cumulative coding rate (10^8 random trials)

On a $<2^4,2^4,2^4,2^4,2^4,2^4+1>/16$ code, using the same method an average number of 10.87 writes was achieved, the 10^{th} write successful in 84% of the cases.
Two practical examples illustrating how to build an efficient memory with the latter code as base element are provided on [WOM page]. In cyclical storage applications (e.g. security video) a rate of above 2.5 can be achieved with it.

Conclusion

A number of techniques were presented that allow to find WOM codes of relatively small size. Most of the codes (except some of the larger ones using coset coding) are small enough to a allow a fast table based encoding and decoding process, while maintaining a high coding rate. Some practical results with high fixed rate were obtained using symbol counts that are a power of 2, which would certainly help the efficient implementation of systems using these codes. Detailed results including decoding maps, parity check matrices and sample programs have been published on dedicated pages

http://users.telenet.be/bertdobbelaere/WOM

The work related to heuristics is far from over in this field. A large number of challenges remain, of which a few will be discussed below.

Future work

Of particular interest is the improvement of the score functions, which currently provide a fair indication of the solution ‘quality’, but offer still a very weak distance metric to better solutions. A similar statement holds for the set of allowed mutations: the mutation types (‘point mutations’ and ‘swaps’) and probabilities were empirically determined, but there is no theoretical base explaining why this choice would give better results than some other mutation types that can be thought of. It is very likely that the convergence to a high quality solution can be made to happen much faster, and in many cases even possible, by applying subtle changes to the score function and/or the allowed set of mutations. Also, I only focused my solution search so far on hill climbing. Alternative approaches like simulated annealing have not been investigated.
Another point of attention is the success rate of the applied mutations. The vast majority of the attempted mutations are “stillborns”, having no change at all to lead to an improvement, but yet having consumed considerable resources to allow the mere detection of their

unfitness. Rules of thumb for quicker detection of chance-less mutations can considerably improve the convergence speed.

Acknowledgements

Special thanks to Dr. Oded Margalit, maintainer of the "*Ponder This*" puzzle pages hosted by IBM research, for triggering my interest in this problem, and to Prof. Adi Shamir for encouraging me to publish my method and results.

The work I performed on WOM codes is a personal initiative, and does not relate to any remunerated activity.

About the author

Bert Dobbelaere works as a software architect in the space industry and holds an MSc in Electrical Engineering from the university of Ghent. He has a broad interest in science and mathematics, including puzzles. Author can be contacted using mail address below title.

References

[RS82] R. L. Rivest and A. Shamir, "How to reuse a write-once memory" Inf. Control, vol. 55, no. 1–3, pp. 1–19, Dec. 1982.

[CGM86] G.D. Cohen, P. Godlewski, and F. Merkx, "Linear binary code for write-once memories," IEEE Trans. Inform. Theory, vol. 32, no. 5, pp. 697–700, September 1986.

[YKS+10] E. Yaakobi, S. Kayser, P. H. Siegel, A. Vardy, and J. K. Wolf. "Efficient two-write wom-codes." Proceedings of IEEE Information Theory Workshop, Dublin, Ireland, 2010.

[YKS+12] E. Yaakobi, S. Kayser, P. H. Siegel, A. Vardy, and J. K. Wolf "Codes for Write-Once Memories", IEEE Transactions on Information Theory, Vol. 58, No. 9, Sep. 2012

[YS14] E. Yaakobi, A. Shpilka, "High sum-rate three-write and non-binary WOM codes" IEEE Transactions on Information Theory (Volume: 60, Issue: 11, Nov. 2014)

[WJ10] Y. Wu and A. Jiang, "Position modulation code for rewriting write-once memories," IEEE Trans. Inform. Theory, October 2010.

[WOM page] http://users.telenet.be/bertdobbelaere/WOM