Noname manuscript No.
(will be inserted by the editor)

Balanced Gray Codes for Reduction of Bit-Flips in Phase Change Memories

Arockia David Roy Kulandai · Stella J · John Rose · Thomas Schwarz

Received: date / Accepted: date

Abstract Phase Change Memories combine byte addressability, non-volatility, and low energy consumption with densities comparable to storage devices and access speeds comparable to random access memory. This qualifies PCM as a successor technology to both DRAM and SSD. The only disadvantage is limited endurance (though still orders of magnitude higher than for Flash memory). Since PCM consume energy only when actively reading and writing and writing consumes much more energy than reading, reducing "bitflip pressure" is important not only in order to stretch endurance, but also to save energy.

We present two related contributions to relieve bitflip pressure. Many data structures use status or dirty bits that over the lifetime of a data structure such as a key value store can flip numerous times. We use balanced Gray codes to distribute the changes of a dirty bit across a whole byte. Our second construction offers a way to implement counters using Gray codes that have close to one bit-flip per increment.

Keywords Phase Change Memories, Bit-flip reduction, Balanced Gray codes

1 Introduction

Phase Change Memories (PCM) are one of several competing technologies with the potential to drastically change Computer Architecture. They combine the speed and byte addressability of Dynamic Random Access Memories (DRAM) with the non-volatility, storage density, storage capacity, and the low costs per

David Roy, T. Schwarz
Marquette University, Milwaukee, Wisconsin, USA,
E-mail: david.roy@marquette.edu, thomas.schwarz@marquette.edu

John Rose, Stella J
Xavier Institute of Engineering, Mahim (West), Mumbai, India,
E-mail: stella@xavier.ac.in, johnrose@xavier.ac.in

Gigabyte of Solid State or Hard Disk Drives (SSD/HDD). The first versions of PCM are already on the market, but specialized to replace (or augment) either DRAM or SSD. Their only draw-back is limited endurance, even though their endurance is still orders of decadic magnitude larger than that of SSD.

To put endurance in perspective, assume we are writing to memory at a rate of 50 GB/sec, close to the current theoretical bandwidth of the data bus. Assume further that we are using the system at this rate for 10 years. Since there are only 3.15576×10^8 seconds in 10 years, we would write a total of 1.57788×10^{10} GB of data during a decade. There is currently a 1.5 TB PCM on the market (albeit only as an SSD and not as a memory replacement). A 1.5 TB PCM would see its bytes overwritten some 1.05×10^7 times, a number that is well within the endurance limits of such a future device. The simple, but important expedient of only overwriting bits that actually have to change (Data Comparison Write DCW [17]) additionally halves the number of overwrites of individual bits. This naive calculation assumes of course perfect load balancing. Without perfect load balancing, the systems community needs to find ways to prevent hot bits from being over-used. Here, we make a small contribution by lowering the bit flips needed for a "dirty bit", used in many lock-free data structures, and of counters. Finally, we investigate whether our solution based on balanced Gray codes [2][10] [14] [15][16] would also be useful for integer addition. We envision a close future where the bulk of memory in computer systems consists of PCM or one of its faster rival technologies, while caches have become even larger. We argue for the importance of reducing bitflips in data structures with a number of reasons.

1. Some data structures, for example those used in the file system, will perdure for the lifetime of a system, and are dangerous to move by a load balancer, which identifies heavily written blocks and swaps them for less often written blocks.
2. Some parts of a data structure will be accessed many times, resulting in "hot" regions with a large number of bit-flips. An example would be the root node of a B-tree.
3. Overusing a single bit in a block should force the load balancer to swap out the complete block, but a load balancer will not be able to track the use of each bit and would therefore be unable to diagnose the problem.
4. PCM use energy only for read and write operations. The costs of overwrites dominates its energy consumption. The less bitflips a data structure has, the less energy is spent on its maintenance. As CPU operations are fast compared to memory operations, we can trade a few additional instruction executions for fewer bitflips.
5. Not all PCM memories will be large. Sensors and embedded systems will be dimensioned as small and as cheap as possible. A combination of steady updates with a small memory can stress endurance.

Our work is speculative, based on a vision of a future that might never happen. If this future happens and if our contributions become reality, they will need support either in hardware (such as translating nibbles and bytes

from the usual integer representation to Gray codes) or at the system level (such as implementing a counter class). Such changes will not come quickly, but if, and this is a big if, if PCM or one of its competing non-volatile storage technologies unify memory and storage, then they will need support at either the OS or the CPU architecture level and probably both.

The remainder of the paper is organized as follows. We first review related work on extending the longevity of PCM and similar non-volatile memories. The third section gives an overview of balanced and other Gray codes. In Section 4, we give our constructions first for implementing a dirty bit and then for implementing counters. We also prove experimentally that balanced Gray codes will not be able to lower bitflips for addition.

2 Related Work

Bit flip pressure can be alleviated at all levels of the storage hierarchy. At the PCM architecture level, the fundamental contribution avoids superfluous writes. Lee *et al.* and Yang *et al.* both observed that we only need to overwrite cells whose content has changed (Partial Write / Data Comparison and Write) [11], [17]. We can safely assume that this proposal is embedded in all current and future PCM devices.

A next group of proposals trades off *ad hoc* encoding with additional bits to be stored. Cho and Lee's *Flip-N-Write* adds a bit cell that determines whether a word is to be read as is or inverted [5]. Palangappa and Mohanram's *Adaptive Flip-N-Write* also stores some words in reversed form [13]. They combine their scheme with light-weigth compression proposed by Alameldeen and Wood and based on recognizing seven frequent patterns for words in memory [1]. Jalili and Sarbazi-Azad observe that bit-flips are not uniformly distributed and only use Flip-N-Write for hot locations. Their scheme, *Captopril*, identifies the hot bits in a block and stores this information as metadata with the block [8].

Just minimizing the number of bit-flips can backfire if the scheme just concentrates bit-flips into a hot set, because frequently written cells could then reach their limit of endurance before the economic lifespan of the system. *Flip-Min* by Jacobvitz, Calderbank and Sorin spreads out bitflips by using coset coding and only write the coset vector with the minimum number of bitflips [7]. Maddah and colleagues propose *Cost-Aware Flip Optimization* (CAFO), which organizes bits in a matrix with auxiliary bits for each row and column indicated whether a cell content is read as is or in inverted form [12]. Han *et al.* use a hardware shuffle to distribute bitflips evenly over a byte or a word [9].

Bittman's work comes closest to ours, as he and his colleagues alleviate bit flip pressure at the systems and data structure level. Besides showing that relatively minute changes in the design of a data structure can yield unexpected bitflip savings, they argue that common structures such as the OS stack should be redesigned [3], [4].

Table 1 Recursive Gray Code for four bits.

Code	Binary	Hex	Bit flipped	Code	Binary	Hex	Bit flipped
r_0	0000	0	3	r_8	1100	c	3
r_1	0001	1	0	r_9	1101	d	0
r_2	0011	3	1	r_{10}	1111	f	1
r_3	0010	2	0	r_{11}	1110	e	0
r_4	0110	6	2	r_{12}	1010	a	2
r_5	0111	7	0	r_{13}	1011	b	0
r_6	0101	5	1	r_{14}	1001	9	1
r_7	0100	4	0	r_{15}	1000	8	0
r_{16}	0000	0	3				
	⋮				⋮		

3 Balanced Gray codes

A Gray code on n bits orders all the 2^n binary strings of length n in such a way that all consecutive strings differ by exactly one bit. Here we are only interested in codes such that the last and first element of the string differs also in exactly one bit. These are more properly called Gray cycles. Gray codes can be interpreted as Hamiltonian circuits on the edges of an n dimensional unit cube.

Probably the best-known Gray code is the Binary Reflected Gray Code (BRGC) presented in Table 1 for length 4. The second column represents the four-digit binary code, the next column the equivalent hexadecimal digit, and the last column gives the position of the single bit flipped between the current code and its predecessor (with 0 for the least significant bit and 3 for the most significant bit). For this code, the least significant bit is flipped eight times whereas the most significant bit is flipped only twice.

Each Gray code is given by the sequence of bits that are flipped, called the *delta sequence* [10] and the first element, customarily 0. In the case of the BRGC, the delta sequence is

$$\delta_{\mathrm{BRGC}} = (0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 3).$$

If we start with a different element or if we apply a permutation on the number representing the bit flipped, then we obtain a different, but equivalent Gray code. For example, if we start with 0111, and apply the same sequence of bits flipped, we obtain a code 0110, 0100, 0101, 0001, ... The delta sequence also allows us to store a Gray code in more compact form, since the size of the elements of the delta sequence is the binary logarithm of the number of different elements. Another way to create an equivalent Gray code is to permute the integers in the delta sequence. The delta sequence $(\delta_i)_{i\in \mathtt{Z}_{2^n}}$ would be replaced by $(\pi(\delta_i))_{i\in \mathtt{Z}_{2^n}}$ with π a permutation of $\{1, \ldots, n\}$.

The Gray code in Table 1 gets its name because of its typical construction. For $n = 1$, there is only one Gray code with bit flip sequence $\Gamma_1 = (0)$, i.e. with code sequence $0 \to 1 \to 0 \ldots$. For $n = 2$, we start with Γ_1, insert 1, and then

Table 2 All balanced Gray codes on four bits starting with 0

```
0,1,3,2,6,7,15,11,9,13,5,4,12,14,10,8
0,1,3,2,6,4,12,14,10,11,15,7,5,13,9,8
0,1,3,2,6,14,15,7,5,13,9,11,10,8,12,4
0,1,3,2,6,14,10,11,9,13,15,7,5,4,12,8
0,1,3,2,6,14,10,8,9,11,15,7,5,13,12,4
0,1,3,2,10,11,15,7,5,13,9,8,12,14,6,4
0,1,3,2,10,8,12,14,6,7,15,11,9,13,5,4
0,1,3,2,10,14,15,11,9,13,5,7,6,4,12,8
0,1,3,2,10,14,6,7,5,13,15,11,9,8,12,4
0,1,3,2,10,14,6,4,5,7,15,11,9,13,12,8
0,1,3,7,6,2,10,14,12,4,5,13,15,11,9,8
0,1,3,7,6,14,15,11,9,13,5,4,12,8,10,2
0,1,3,7,5,4,12,13,9,11,15,14,6,2,10,8
0,1,3,7,5,4,12,14,6,2,10,11,15,13,9,8
0,1,3,7,5,13,12,4,6,2,10,14,15,11,9,8
0,1,3,7,5,13,15,14,6,2,10,11,9,8,12,4
0,1,3,7,5,13,9,8,12,4,6,14,15,11,10,2
0,1,3,7,15,13,5,4,12,8,9,11,10,14,6,2
0,1,3,7,15,11,10,8,9,13,5,4,12,14,6,2
0,1,3,7,15,11,9,8,10,2,6,14,12,13,5,4
0,1,3,7,15,11,9,8,12,13,5,4,6,14,10,2
```

repeat Γ_1. This gives bit flip sequence $\Gamma_2 = (0, 1, 0)$ and Gray code sequence $00, 01, 11, 10$. Let the dot . denote concatenation of sequences. For the general case we then have

$$\Gamma_{n+1} = \Gamma_n.(n-1).\Gamma_n.$$

Thus, $\Gamma_3 = (0, 1, 0, 2, 0, 1, 0)$. For Γ_4, the delta sequence has 3 between two copies of Γ_3 resulting in the delta sequence above and the Gray code of Table 1.

If we use the binary reflected Gray code to increment a value in $0, 1, 2, \ldots, 15$ by one before rolling back to 0000, we flip only one bit at a time, but we do not flip each bit equally often. A Gray code is called *balanced* if it distributes the number of bit flips evenly over all bits. Let x denote this number. If a Gray code over n digits is balanced and has x bit flips in each bit, then $n \cdot x = 2^n$, which implies that both n and x are powers of two.

We look for balanced Gray codes for $n = 4$. We can assume the start to be zero and the first bit to be flipped to be 0. The next bit to be fliped cannot be 0, so we can assume it to be 1. This means, that we are looking for balanced Gray codes that start out with $0000, 0001, 0010$. For the next step, we can make a case distinction, namely whether the next bit to be flipped is among those already flipped or a new one. In the first case, the only possibility is to flip 0, in the second case, up to permutation of the set of bits flipped, we can assume that this is bit 2. This means that if a balanced Gray code exists, then it must begin with 0000, 0001, 0011, 0010 or with 0000, 0001, 0011, 0111. In the first case, we know that we have to now flip a different bit, and up to equivalence, this has to be bit three, Therefore, this family of balanced Gray codes starts out with 0000, 0001, 0011, 0010, 0110. We can now use a computer

Table 3 Balanced Gray code on eight bits [15].

```
00,01,03,02,06,0e,0a,0b,09,0d,0f,07,05,04,0c,08,
18,1c,14,15,17,1f,3f,37,35,34,3c,38,28,2c,24,25,
27,2f,2d,29,39,3d,1d,19,1b,3b,2b,2a,3a,1a,1e,16,
36,3e,2e,26,22,32,12,13,33,23,21,31,11,10,30,20,
60,70,50,51,71,61,63,73,53,52,72,62,66,6e,7e,76,
56,5e,5a,7a,6a,6b,eb,ea,fa,da,de,d6,f6,fe,ee,e6,
e2,f2,d2,d3,f3,e3,e1,f1,d1,d0,f0,e0,a0,b0,90,91,
b1,a1,a3,b3,93,92,b2,a2,a6,ae,be,b6,96,9e,9a,ba,
aa,ab,bb,9b,99,9d,dd,d9,db,fb,7b,5b,59,5d,7d,79,
f9,fd,bd,b9,a9,e9,69,6d,6f,67,65,64,e4,e5,e7,ef,
ed,ad,af,a7,a5,a4,ac,ec,6c,68,e8,a8,b8,f8,78,7c,
fc,bc,b4,b5,b7,f7,f5,f4,74,75,77,7f,ff,bf,9f,df,
5f,57,55,54,d4,d5,d7,97,95,94,9c,dc,5c,58,d8,98,
88,c8,48,4c,cc,8c,84,c4,44,45,c5,85,87,c7,47,4f,
cf,8f,8d,cd,4d,49,c9,89,8b,cb,4b,4a,ca,8a,8e,ce,
4e,46,c6,86,82,c2,42,43,c3,83,81,c1,41,40,c0,80
```

search to find all balanced Gray codes of length 4, given in Table 2, subject to these restrictions.

The first general existence proof for balanced Gray codes comes from Wagner and West for a number of digits that are powers of 2 [16] and extended to all length by Bhat and Savage [2]. Table 3 shows a balanced Gray code for length 8. We find them less useful in our context as translating an integer $i \in \{0, 1, \ldots, 255\}$ to the i^{th} element of the code or calculating the next element in the Gray code – perhaps via the delta sequence – leads to larger circuits or more space consuming software implementations.

4 Implementing a Dirty Bit

Dirty bits are a common tool in designing lock-free data structures. The use of a *Compare-And-Swap* (CAS) or similar atomic instruction insures that updates to the dirty bit will be written directly to main memory bypassing the caches and invalidating their entries. Since there are only $10 \times 365.25 \times 24 \times 60 \approx 5 \times 10^6$ minutes in ten years, flipping a dirty bit regularly, even if it were for a data structure that persists over the lifetime of the system, is usually not a large issue. However, this consideration does not apply if the dirty bit belongs to a system data structure, e.g. for memory management, that is changed more often, since we then would be approaching the 10^7 to 10^8 endurance of PCM. For this rare, but important cases, we present here a solution to spread out the dirty bit over several bits.

Our first solution combines the dirty bit with a counter. This can be useful to evade the ABA problem [6]. Abstractly, it creates a data structure with an internal value field (such as a nibble, a byte or several bytes) that implements a *flip* function and an *evaluate* function which returns the value of the dirty bit, that is, either zero or one. Our first solution uses a balanced Gray code to store the value in a nibble or a byte. The evaluate method returns the parity of

```
class DirtyBit:
  nextvector = [ ... ]
  def __init__(self):
     self.value = 0
  def getValue(self):
     ...
  def next(self):
    right = self.value&1
    index = self.value>>1
    info = DirtyBit.nextvector[index]
    bit = info>>4 if right else info&0xf
    self.value^=0x1<<bit
```

Fig. 1 Python pseudo-code for the next function of a dirty bit implementation.

the value field. The flip operation just advances the Gray code by one. Since two consecutive words in the Gray code differ by one bit, the parity of the value field is guaranteed to change.

To implement the advancement in software, we can use two methods. First, for all possible values, we store the next value in an array. If the values are bytes, there are 256 values, so that we need an array of 256 bytes. This array is read-only and can easily fit into a cache. At the cost of more calculation, we can save half the space by only adding the integer representation of the bit to be flipped. Since the bit number only needs 4b storage, we can store the information in half a byte. We therefore create a 128B array such that the left half of the byte at index i gives the bit to flip when the current value is $2i$ and the right half of the byte at index i gives the bit to flip when the current value is $2i + 1$. To extract it (Figure 1), we divide the current value of the dirty bit into the trailing binary digit and into the current value shifted to the right. The first number tells us whether the desired information is in the left or in the right half of the byte, the second number gives us the address of the byte in the lookup array (DirtyBit.nextvector).

If the value is a nibble, we only need 8 bytes to encode the next array, which consists of 16 half-bytes. We can also use the delta sequence, which consists of 16 elements that can be encoded in two bits, for a total of 4 bytes.

The calculations are performed in CPU and do not need to write anything but the new value of the dirty bit. Since the code can be shared by all processes, we can even expect it to be always cache resident.

If we only use a single nibble, then it is very easy to implement the next function in hardware.

To determine the parity, we can also use a hardware solution, but software implementations are also readily available. One possibility would be a 256 bit look-up array of size 32B.

When we initialize a dirty bit in a byte, we do not have to change the value of the byte about half the time, namely when it already has the correct parity. If this is not the case, then we simply flip the dirty bit.

If we are only interested in flipping a dirty bit, the use of the balanced Gray code is overkill. An alternative is using a sequence like 000000000000,

```
if x&0x8000:
    x = (x<<1)&0xffff
else:
    x = ((x<<1)|0x01)&0xffff
```

Fig. 2 Python pseudo-code for implementing a dirty bit in 16 actual bits.

000000000001, 000000000011, 000000000111, ..., 011111111111, 111111111111, 111111111110, 111111111100, ..., 100000000000, 000000000000. This can also be implemented in software, as is examplified by the code fragment given in Figure 2. Depending on the most significant bit, we either use a left shift or a left shift combined with setting the least significant bit. We can trade clarity for terseness by replacing the if-else construct with additional bit-wise operations.

5 Implementing a Counter

While the setting of our dirty bit can be simply evaluated by calculating the parity of the value field, we often need to calculate the value of a counter. This is not always necessary, as sometimes we just need to count to a given value. We represent a number as usual in binary format. We then break this number up into nibbles and replace each nibble i with the i^{th} element of a Gray code $(\beta_i)_{i=0,\ldots 15}$ such as $B_4 = (0,\ 1, 3, 2, 6, 4, c, e, a, b, f, 7, 5, d, 9,\ 8)$. We assume that this translation happens in hardware. We can think of this encoding as encoding integers with base 16, but using a different representation for hexadecimal letters.

If we start at 0 and increment a n-bit counter until we roll over to 0 again, we flip the least significant digit 2^n times, the second least significant digit 2^{n-1} times, and so on. The most significant digit is flipped only twice. The total number of bit-flips is

$$\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$$

times or per increment operation $2 - 2^{1-n}$ times. If we however use our encoding, we flip one bit in the least significant hexadecimal digit once per increment or a total of 2^n times, one bit in the second least significant hexadecimal digit every 16 increments or 2^{n-4} times, and one bit in the most significant hexadecimal digit 2^4 times. The total number of bit-flips is

$$\sum_{i=1}^{\frac{n}{4}} 2^{4i} = \frac{16}{15}(2^n - 1)$$

and the average per increment operation is now $\frac{1}{15}2^{4-n}\,(2^n - 1)$, which is only 8/15 of the previous numbers and close to the optimal number of 1 bitflip per increment.

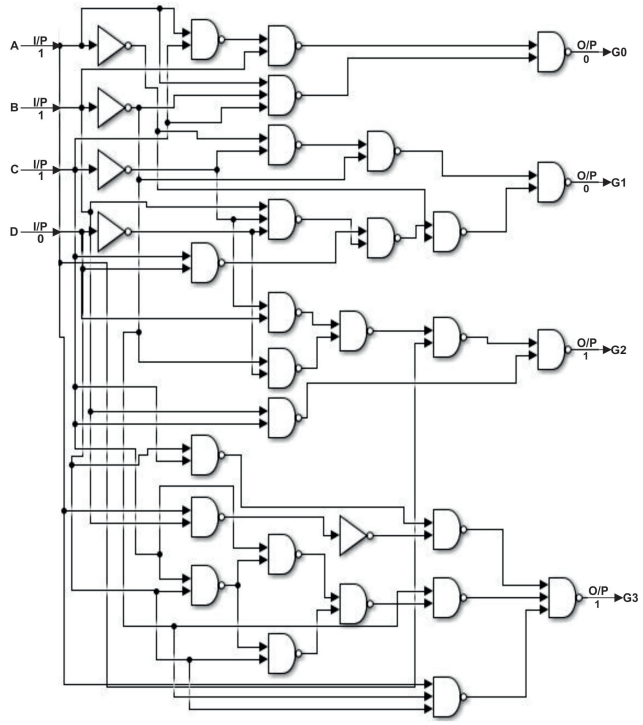

Fig. 3 Hardware implementation of a translation from 4-bit unsigned integer to Gray code.

If instead we represent the integer in base 256 and then use a balanced Gray code on eight binary digits, the number of bitflips goes down to 128/255, that is, we improve from a savings of 46.67% of bitflips to 49.80% of bitflips.

5.1 Implementation

Somewhere on the way between the CPU and PCM memory, we need to transfer between the normal integer representation and the representation using a Gray code. In software, this is best done using a loopup table. Such a lookup table is likely to reside in cache, but in any case will not cause any writes to PCM. We can also look at hardware implementations. For this, we break each integer into nibbles, e.g. on a 64b architecture into 16 nibbles. For each nibble, we then have circuits translating between the standard unsigned integer and the Gray code. Figure 3 gives an example. The circuit implements the balanced Gray code $(0, 1, 5, 4, 12, 8, 10, 14, 6, 7, 15, 13, 9, 11, 3, 2)$ as a translation circuitry. If $ABCD$ is a nibble, then the corresponding element $(G_0G_1G_2G_3)$ of the Gray code is $G_0 = B(\bar{A}+\bar{C})+A\bar{B}C$, $G_1 = \bar{B}(A+C)+\bar{A}(CD+BC\bar{D})$, $G_2 = BC+A(\bar{C}D+B\bar{D})$, $G_3 = AB(\bar{C}+\bar{D})+\bar{B}(C\oplus D)+A\bar{B}D$. This can be

Table 4 Good four-bit balanced Gray codes

Group A

```
[0, 1, 3, 2, 6, 4, 12, 14, 10, 11, 15, 7, 5, 13, 9, 8]
[0, 1, 3, 2, 10, 8, 12, 14, 6, 7, 15, 11, 9, 13, 5, 4]
[0, 1, 5, 4, 12, 8, 10, 14, 6, 7, 15, 13, 9, 11, 3, 2]
[0, 1, 9, 8, 10, 2, 6, 14, 12, 13, 15, 11, 3, 7, 5, 4]
[0, 1, 9, 8, 12, 4, 6, 14, 10, 11, 15, 13, 5, 7, 3, 2]
[0, 2, 3, 1, 5, 4, 12, 13, 9, 11, 15, 7, 6, 14, 10, 8]
[0, 2, 6, 4, 5, 1, 9, 13, 12, 14, 15, 7, 3, 11, 10, 8]
[0, 2, 6, 4, 12, 8, 9, 13, 5, 7, 15, 14, 10, 11, 3, 1]
[0, 2, 10, 8, 9, 1, 5, 13, 12, 14, 15, 11, 3, 7, 6, 4]
[0, 4, 6, 2, 10, 8, 9, 11, 3, 7, 15, 14, 12, 13, 5, 1]
[0, 4, 5, 1, 9, 8, 10, 11, 3, 7, 15, 13, 12, 14, 6, 2]
[0, 4, 12, 8, 10, 2, 3, 11, 9, 13, 15, 14, 6, 7, 5, 1]
[0, 4, 5, 1, 3, 2, 10, 11, 9, 13, 15, 7, 6, 14, 12, 8]
[0, 8, 9, 1, 5, 4, 6, 7, 3, 11, 15, 13, 12, 14, 10, 2]
[0, 8, 10, 2, 6, 4, 5, 7, 3, 11, 15, 14, 12, 13, 9, 1]
[0, 8, 12, 4, 5, 1, 3, 7, 6, 14, 15, 13, 9, 11, 10, 2]
```

Group B

```
[0, 1, 3, 2, 6, 14, 10, 8, 9, 11, 15, 7, 5, 13, 12, 4]
[0, 1, 3, 2, 10, 14, 6, 4, 5, 7, 15, 11, 9, 13, 12, 8]
[0, 1, 9, 8, 10, 14, 12, 4, 5, 13, 15, 11, 3, 7, 6, 2]
[0, 1, 9, 8, 12, 14, 10, 2, 3, 11, 15, 13, 5, 7, 6, 4]
[0, 2, 3, 1, 5, 13, 9, 8, 10, 11, 15, 7, 6, 14, 12, 4]
[0, 2, 3, 1, 9, 13, 5, 4, 6, 7, 15, 11, 10, 14, 12, 8]
[0, 2, 6, 4, 12, 13, 5, 1, 3, 7, 15, 14, 10, 11, 9, 8]
[0, 2, 10, 8, 12, 13, 9, 1, 3, 11, 15, 14, 6, 7, 5, 4]
[0, 4, 12, 8, 9, 11, 10, 2, 6, 14, 15, 13, 5, 7, 3, 1]
[0, 4, 6, 2, 10, 11, 3, 1, 5, 7, 15, 14, 12, 13, 9, 8]
[0, 4, 6, 2, 3, 11, 10, 8, 12, 14, 15, 7, 5, 13, 9, 1]
[0, 4, 5, 1, 9, 11, 3, 2, 6, 7, 15, 13, 12, 14, 10, 8]
[0, 8, 9, 1, 3, 7, 5, 4, 12, 13, 15, 11, 10, 14, 6, 2]
[0, 8, 10, 2, 6, 7, 3, 1, 9, 11, 15, 14, 12, 13, 5, 4]
[0, 8, 12, 4, 5, 7, 6, 2, 10, 14, 15, 13, 9, 11, 3, 1]
[0, 8, 12, 4, 6, 7, 5, 1, 9, 13, 15, 14, 10, 11, 3, 2]
```

implemented with 32 NAND gates. For a number of other codes that also save a few bitflips for addition (see below) in Table 4, we found implementations with 32 NAND gates.

5.2 Addition

Since encodings based on Gray codes lower the number of bitflips when employing a counter, we can ask whether the same ploy also works for addition. With a large cache, numerical results will only overwrite operands after involved calculations and encoding is unlikely to accrue savings, but this might not be the case for an embedded system with a small PCM and puny processor.

We experimented by determining the expected number of bitflips for addition under various encodings. In the absence of a better model, we assume that summands are according to a Zipfian distribution in a range $[0, 2^m]$. We calculated the expected number of bitflips in an immediate addition `x += y`, where `x` and `y` are taken from this range. As possible candidates for encodings we used all balanced Gray codes in Table 2 including those obtained by slicing

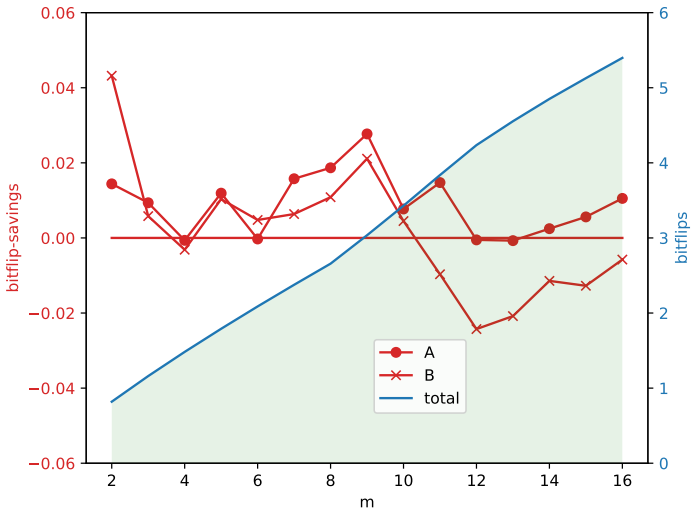

Fig. 4 Expected number of bitflips when adding using the traditional encoding (blue) and savings when using Codes from Groups A and B (red). The x-axis gives the range of the operands as $[0, 2^m]$.

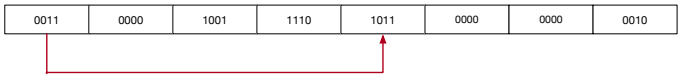

Fig. 5 A load balanced counter with raw value 0010 0000 1001 1110 1011 or $15 \cdot 16^4 + 0 \cdot 16^3 + 12 \cdot 16^2 + 7 \cdot 16 + 13 = 986237$ if we use the 4-digit Gray code implemented in Figure 3. The first nibble points to the least significant digit of the counter value.

the delta sequence into two parts and rejoining them in reverse order. Our search gave us two groups, A and B in Table 4 that reasonably consistently gave savings. The savings as well as the total number of expected bitflips using the normal binary encoding are given in Figure 4. As the savings are not consistent and on the order of 1%, the results are not encouraging.

5.3 Load balancing

Using our or any of the many other schemes to save bitflips can backfire if we manage to concentrate a lesser number of bitflips in relatively few cells, which might reach the limit of their endurance before the end of the economic lifespan of the system.

A number of hardware based proposals [7], [9], [12] are effective in doing so, but it is unclear to us whether their hardware overhead will make them attractive to device manufacturers. History has shown that the cost-savings of large numbers mitigate optimizations for special cases. We therefore need a software solution. One possibility is to use the first nibble of a 32b word

for load balancing, limiting of course the maximum value to which we can count to 2^{28}. The first nibble contains a value between zero and seven, which is interpreted as a pointer to the least significant nibble of the counter (see Figure 5). We advance the offset whenever the three least significant nibbles roll over zero, resulting in a burst of bitflips. When we advance, we have to move seven nibbles, however, three of them we know to be zero. Of the seven nibbles overwritten, thus two are overwriting a zero nibble with a zero nibble, so that with Data Comparison Write, we overwrite 5 nibbles, leading on average to 20 bit flips. For a counter that eventually reaches a large number N (less than 2^{24}), we therefore expect

$$\frac{20N}{16^3} = 0.00488N$$

additional bitflips caused by our load balancing scheme. This is an increase of less than half a percent.

To achieve load balancing for counters that are not heavily used, we can load the first nibble with a random number when initializing a counter.

6 Conclusions

PCM is one of a class of non-volatile storage technologies that is capable of uniting the role of memory and storage in current computer architectures. Of the many competing technologies, it is the first to be on the market in terabyte size. While it is still undergoing rapid technological development and while we still do not know whether it is going to be the winner in the current race, now is the time to think about its uses and the technological adaptions that are needed. Like many of its competitors, PCM have limited endurance. Therefore, reducing bitflip pressure is important, in particular if the size of the PCM is limited. An additional reason to limit the number of bitflips lies in the concentration of energy consumption of PCMs in the writes.

Here, we investigated situations where the limited endurance of PCM is an issue. We identified two issues. First, the "dirty bit" that is part of many important data structures, including system data structures that lasts for the lifetime of the system. Second, a counter such as might be implemented in PCM itself in order to gather the statistics on writes to a block so that load balancing becomes possible.

In both cases, we showed that using balanced Gray codes allows us to save bitflips and even more importantly, to distribute the bitflip load equally over the bits. In addition, we proposed a simple way of distributing the dirty bit over many bits.

References

1. Alaa Alameldeen and David Wood: Frequent pattern compression: A significance-based compression scheme for L2 caches, Technical Report, University of Wisconsin-Madison, 2004.

2. Girish Bhat, Carla Savage: Balanced Gray Codes, The electronic journal of combinatorics, vol. 3, R25, 1996.
3. Daniel Bittman *et al.*: Designing Data STructures to Minimize Bit Flips on NVM, 2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium, pp. 85–90, 2018.
4. Daniel Bittman, Darrell Long, Peter Alvaro, and Ethan Miller: Optimizing systems for byte-addressable NVM by reducing bit flipping, 17th Conference on File and Storage Technologies, USENIX, 2019.
5. Sangyeun Cho and Hyunjin Lee: Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 347-357. 2009.
6. Damian Dechev, Peter Pierkelbauer, and Bjarne Stroustrup: Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pp. 185-192, 2010.
7. Adam Jacobvitz, Robert Calderbank, and Daniel Sorin: Coset coding to extend the lifetime of memory, IEEE 19th International Symposium on High Performance Computer Architecture, pp. 222-233, 2013.
8. Majid Jalili and Hamid Sarbazi-Azad: Captopril: Reducing the pressure of bit flips on hot locations in non-volatile main memories, Conference on Design, Automation & Test in Europe, DATE '16, 2016.
9. Miseon Han, Youngsun Hun, Seon wook Kim, Hokyoon Lee, and Il Park: Content-aware bit shuffling for maximizing PCM endurance, ACM Transactions on Design Automation of Electronic Systems, vol. 22(3), pp. 1–26, 2017.
10. Donald Knuth: **The art of computer programming: Generating all combinations and permutations**, vol. 4, Fac. 2, Addison Wesley, 2005.
11. Benjamin Lee, Engin Ipek, Engin, Onur Mutlu, and Doug Burger: Architecting phase change memory as a scalable DRAM alternative, ACM SIGARCH Computer Architecture News, vol. 37(3), pp. 2–13, 2009.
12. Rakan Maddah, Seyed Seyedzadeh, and Rami Melhem: CAFO: Cost aware flip optimization for asymmetric memories, IEEE 21st International Symposium on High Performance Computer Architecture, pp. 320-330, 2015.
13. Poovaiah Palangappa and Kartik Mohanram: Flip-Mirror-Rotate: An architecture for bit-write reduction and wear leveling in non-volatile memories, Great Lakes Symposium on VLSI, pp. 221–224, ACM, 2015.
14. Carla Savage: A survey of combinatorial Gray codes, SIAM review, vol. 39(4), pp. 605-629, 1997.
15. I Nengah Suparta: Counting sequences, Gray codes and Lexicodes, Ph.D. thesis, TU Delft, 2006.
16. D. Wagner and J. West: Construction of uniform Gray codes, Congressus Numerantium, vol. 80, pp. 217–223, 1991.
17. B. Yang, J. Lee, J. Kim, J. Cho, S. Lee, and B. Yu: A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme, 2007 IEEE International Symposium on Circuits and Systems, pp. 3014-3017, 2007.