Supplement to ZombieMemory: Extending Memory Lifetime by Reviving Dead Blocks

John D. Davis, Karin Strauss, Parikshit Gopalan, Mark Manasse, Sergey Yekhanin
Microsoft Research
{john.d,kstrauss,parik,manasse,yekhanin}@microsoft.com

Abstract

Zombie is an endurance management framework that enables a variety of error correction mechanisms to extend the lifetimes of memories that suffer from in-the-field bit failures, such as wearout in phase-change memory (PCM). Zombie supports both single-level cell (SLC) and multi-level cell (MLC) variants. It extends the lifetime of blocks in working memory pages (primary blocks) by pairing them with spare blocks, i.e., working blocks in pages that have been disabled due to exhaustion of a single block's error correction resources, which would be 'dead' otherwise. Spare blocks adaptively provide error correction resources to primary blocks as failures accumulate over time. This reduces the waste caused by early block failures, making all blocks in discarded pages a useful resource. Even though we use PCM as the target technology, Zombie applies to any memory technology that suffers stuck-at cell failures.

This paper provides supplemental information to [4], which describes the Zombie framework, a combination of two new error correction mechanisms (ZombieXOR for SLC and ZombieMLC for MLC) and the extension of two previously proposed SLC mechanisms (ZombieECP and ZombieERC). We present the read and write algorithms, an analytical model for SLC, detailed discussion of the MLC mechanism, and analysis demonstrating reduced drift-induced soft errors for MLC. This additional information could not fit in the page limitations of [4], demonstrates feasibility of PCM, especially MLC, and supports our results of 58% to 92% improvement in endurance for Zombie SLC memory and an even more impressive 11× to 17× improvement for ZombieMLC, both with performance overheads of only 0.1% when memories using prior error correction mechanisms reach end of life.

1. Introduction

The current technology roadmap shows that scaling DRAM to smaller features [8] is rapidly slowing down. Fortunately, DRAM replacement solutions are emerging [12, 13, 16]. These solutions provide more stable storage and are based on magnetic or physical properties of materials. Phase-change memory (PCM) [1, 3, 5], a resistive memory technology, is already shipping as a NOR-Flash replacement. It is faster, uses less power, and achieves longer lifetimes than Flash.

Using PCM as a DRAM replacement for main memory, however, poses challenges. One such challenge is the endurance of individual bits. While DRAM cells typically support an average of 10^{15} writes over their lifetime, PCM cells last for as little as 10^8 writes on average. Permanent DRAM cell failures are so rare that mechanisms to tolerate them are wasteful: They disable the entire physical page where the failure occurred, which then becomes unavailable for software use. Since PCM cells wear out much faster, using the same approach would quickly disable all PCM pages. Thus, to make PCM a viable alternative for main memory, lifetime-extending mechanisms are crucial for both single-level cell (SLC) and multi-level cell (MLC) PCMs. An additional challenge with MLC PCM is drift: Once written, cell resistance may change over time. This adds complexity to MLC error correction mechanisms because they must tolerate both wearout and drift.

Multiple hardware-only error correction mechanisms tackle wearout by transparently correcting and hiding failures from software layers [7, 11, 14, 15, 19]. Despite significant progress, these mechanisms remain inefficient, wasting a large number of working bits when they can no longer correct errors and thus must disable pages.

To illustrate the opportunity Zombie leverages, Figure 1 shows the average number of bit flips (or writes to a bit) and the average fraction of failed bits when a page must be disabled, for several previously proposed mechanisms that protect SLC PCM. We have added results for our PCM SLC schemes to the graph as well. The fraction of failed bits measures the amount of waste — the lower this fraction, the higher the number of unusable working bits that are wasted. All practical mechanisms waste at least 99% of the bits in a page. Oracle64 and Oracle128 represent ideal mechanisms that correct 64 and

128 bit failures per block, respectively. Even though increasing the error tolerance from 64 to 128 bit failures dramatically reduces bit waste, it is arguably in the diminishing returns region with respect to increasing the number of bit flips. Note that Oracle64 increases memory endurance (68 million flips) by 50% or more compared to other previously proposed mechanisms (SAFER at 44 million flips), representing a significant memory lifetime improvement opportunity that can be realized by the Zombie framework. The opportunity is significantly larger for MLC PCM. We also show our new ZombieSLC mechanisms (Zombie + Error Correcting Points: ZECP, Zombie + Erasure Codes:ZERC, and Zombie + Primary and Spare block XOR: ZXOR) plotted with these other mechanisms demonstrating the significant increase in the percentage of failed memory cells. Our ZombieSLC mechanisms increase the percentage of failed bits in a page by almost 20×.

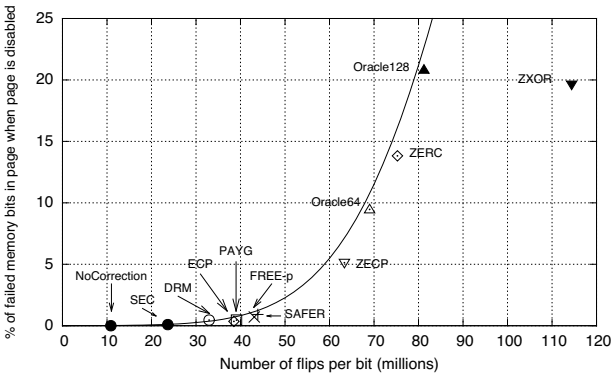

Figure 1: Average number of bit flips (x-axis) and average fraction of failed bits (y-axis) when a page must be disabled for a variety of error correction mechanisms that protect an SLC PCM (represented by markers), assuming an average cell lifetime of 10^8 bit flips (writes) and a 0.25 coefficient of variance. The solid line is the cumulative distribution function of failed memory bits as a function of bit flips.

By leveraging this opportunity, the ZombieSLC mechanisms extend memory lifetimes by 58% to 92%. ZombieMLC achieves even more impressive lifetime extensions — of 11× to 17×— while also tolerating drift. Zombie achieves these longer lifetimes with graceful degradation and at low performance overhead and complexity.

The Zombie framework lets a variety of error correction mechanisms use the abundant working bits in disabled pages to extend the lifetime of pages still in service. Its unifying principle is to pair a block, or even a subblock, sourced from disabled pages (*spare blocks*) with a block in software-visible pages (*primary blocks*), extending the primary block's useful lifetime (and turning them into 'zombies'). Zombie enables on-demand pairing and gradual spare subblock growth, i.e., primary blocks are paired with spare blocks only when they exhaust their own error correction resources and can gradually increase their spare subblock size as additional resources are needed.

This paper provides supplemental information to [4], which originally proposed two new error correction mechanisms, ZombieMLC and ZombieXOR, and extends two existing ones (ZombieECP and ZombieERC) to showcase the Zombie framework. ZombieMLC, as the name suggests, is designed specifically for MLC and, to our knowledge, is the first mechanism to tolerate both drift and stuck-at failures. To demonstrate ZombieMLC feasibility, we provide an analysis of the resistance levels and drift-induced soft errors. The other mechanisms (ZombieXOR, ZombieECP and ZombieERC, collectively called "ZombieSLC mechanisms") tolerate only stuck-at failures and are better suited to SLC PCM.

The rest of this paper is structured as follows. Section 2 presents the read and write (we assume read-write-verify or differential writes) algorithms for PCM-like memories. Section 3 provides an analytical model that matches the simulation results for SLC mechanisms. Section 4 offers more details and examples for the ZombieMLC mechanism and Section 5 evaluates the drift-induced soft error tolerance, making PCM a viable MLC technology. Section 6 concludes our discussion.

2. Algorithms

We present the three algorithms to access PCM memory using Zombie. Algorithm 1 shows how to perform memory reads. We first read the given block position and detect, from its metadata, if it is paired. For paired blocks, we need to read both the primary block and its spare, for unpaired blocks, we only require the intrinsic ECP. If the block is paired, we need to read the spare address and the spare value with ECP. The next step is to decode the block and spare using the selected encoding method (ZombieECP or ZombieXOR). In our evaluation, only one method is used at a time, which simplifies the real algorithm implementation.

We divided writes into two algorithms, depending on the technique the Zombie used. The first, Algorithm 2, implements ZombieXOR and ZombieECP. Implementing ZombieECP is easier as we based our implementation on top of an ECP-enabled memory controller; so it is only necessary to increase the number of ECP bits. For ZombieXOR we start by finding the spare address, which may already be cached. The new spare value ($S' = P\ xor\ P'$) is

Algorithm 1 Memory Read

```
INPUT: A memory block address Addr_P
OUTPUT: The value at memory position Addr_P
MemRead(Address: Addr_P)
P = Read(Addr_P)
if IsPaired(P) then
  Addr_S = GetSpareAddress(Addr_P)
  S = ReadECP(Addr_S)
  if IsAdaptive(P) then
    if IsErasure(P) then
      return ErasureDecoding(P,S)
    else
      return ECPCorection(P,S)
    end if
  else
    return P xor S
  end if
else
  P = ReadECP(Addr_P)
  return P
end if
```

calculated to minimize bit flips in the main block. After verifying S', if S' does not match S, the next step is to calculate the bits that should be written to P (S' *xor* P). If we still have errors, we use the ECP bits in S to correct the aligned errors. When the ECP bits in S are exhausted, we unpair the block and try to find a new pair, restarting MemWrite. Only a small number of unpair/pair are allowed inside the write. If no compatible pair is found, the primary block and its page are placed in the spare pool and the spare block is disabled.

Algorithm 3 implements Memory Write for ZombieERC. As with all pairing methods, it starts by looking up the spare block address. At the same time, the error locations are fetched from the Error Location Cache. If they are not present, the controller does two inverted writes to the memory to detect the stuck-at errors in the primary and spare and updates the cache. After that, the Erasure Code is calculated and written to the memory. If an error happens, the operation is repeated. When the ERC is exhausted, the block is unpaired and paired again with a bigger spare that can hold more stuck-at errors.

3. Analytical Model

We can build an analytic model for Zombie lifetime extension techniques based on the probability p of a write failure occurring uniformly random across the memory. The probability that no errors occur in our memory after a single write to each location is $(1-p)^{512}$. Probability p is typically fairly small, on the order of $10^{-9} \leq p \leq 10^{-8}$.

Algorithm 2 *XOR* Memory Write

```
INPUT: A memory block address Addr_P, the value P
to be written, and the list of spare blocks L.
OUTPUT: none
MemWrite(Address: Addr_P, Block: P)
if IsPaired(P) and !IsAdaptive(P) then
  P' = PreRead(Addr_P)
  Addr_S = GetSpareAddress(Addr_P)
  S = P xor P'
  S' = WriteECP(Addr_S, S)
  if S' ≠ S then
    P' = S' xor P
    WriteOnly(Addr_P, P')
    P' = PosRead(Addr_P)
    if P ≠ (P' xor S') then
      Update ECP bits on S
      if ECP bits exhausted then
        UnPair(Addr_P, Addr_S)
        Append(L, Addr_S)
        FindPair(Addr_P, L)
        MemWrite(Addr_P, P)
      end if
    end if
  end if
else
  WriteECP(Addr_P, P)
end if
```

If we set $N \approx \frac{1}{p}$, after N writes, the probability that no failures have occurred is $(1-\frac{1}{N})^N \approx \frac{1}{e}$, so it is quite likely for some cell to have failed. Now, if we consider kN writes for small k, (typically $0.1 \leq k \leq 20$ with j failures, where j is a number in the single or small double digits and $kN \gg j$), we have no failures with probability $\frac{1}{e^k}$ and j failures with probability $\approx \frac{k^j}{j!e^k}$. Our model computes these binomials more exactly using built-in Excel functions for binomial coefficients based on the error correction characteristics (number of writes per memory operation, expected bit flips, etc.) for ECP and Zombie, as shown below in Figure 2.

Figure 2 is a good approximation of our system, but has some shortcomings. In ZombieXOR, we re-pair primary blocks with spare blocks to find new combinations which do not have aligned errors. This is something not covered in the analytic model. We found empirically that re-pairing significantly extends the lifetime of primary blocks, greatly extending the 50% lifetime plateau. Thus, the analytic framework provides a fast mechanism to compare error correction schemes (Zombie is better than ECP, SAFER and FREE-p, the latter two not shown in Figure 2), but simulation is still required for mechanisms that

Algorithm 3 ZombieERC Memory Write

INPUT: A memory block address $Addr_P$, the value P to be written, the bit error locations E, the codeword C, and the lists (by size: 128 bit, 256 bit, and 512 bit) of spare blocks $L[0:2]$.
OUTPUT: none
MemWrite(Address: $Addr_P$**, Block:** P**)**
if IsPaired(P) **and** IsAdaptive(P) **then**
  P' = PreRead($Addr_P$)
  $Addr_S$ = PairAddress($Addr_P$)
  E = ErrorLocationCache($Addr_P$, $Addr_S$)
  if E == NULL **then**
    $\bar{S}$ = Write($\sim S$)
    $\bar{P'}$ = Write($\sim P'$)
    E = $!(P'$ **xor** $\bar{P'})CAT!(S$ **xor** $\bar{S})$
    UpdateErrorLocationCache($Addr_P$, $Addr_S$, E)
  end if
  C = ErasureEncoding(P', S, E)
  C' = WriteERC($Addr_P$, $Addr_S$, C)
  if $C' \neq C$ **then**
    E' = $!(C'$ **xor** $C)$
    E'' = E **or** E'
    UpdateErrorLocationCache($Addr_P$, $Addr_S$, E'')
    if FailedCodeword(C') **then**
      C = ErasureEncoding(P', S, E'')
      C' = WriteERC($Addr_P$, $Addr_S$, C)
    end if
  end if
  if ERC exhausted **then**
    UnPair($Addr_P$, $Addr_S$)
    Append($L[size++]$, $Addr_S$)
    FindPair($Addr_P$, $L[size]$)
    MemWrite($Addr_P$, P)
  end if
else
  WriteECP($Addr_P$, P)
end if

are hard to model analytically as juxtaposed to Figure 3, where the simulation does spare block repairing.

Noting the differences between the simulations and models, we reflected on Perfect Matching work done initially by Karp and Sipser [9], as improved by Aronson et al. [2], to realize that greedy re-pairing was likely to succeed as long as the number of potential matches was large enough. Combining the estimate of the number of failures, and considering birthday paradox collisions, we were able to predict the probability of successful repairing from the error rate; we then folded this into the analytic model to extend that model, with the resulting expected lifetime now a good match to the simulation results.

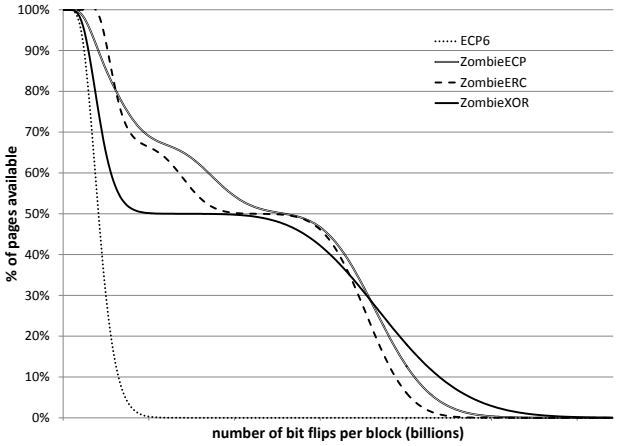

Figure 2: **Analytical predictions of PCM lifetime for ECP, ZombieECP, ZombieERC, and ZombieXOR.**

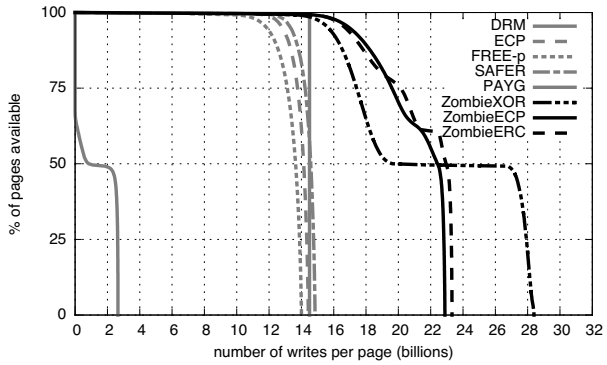

Figure 3: **Simulation predictions of PCM lifetime for ECP, ZombieECP, ZombieERC, and ZombieXOR.**

4. ZombieMLC: Drift-Tolerance plus ECC

ZombieMLC overcomes the challenges of drift and stuck-at cell faults for MLC non-volatile memory by combining rank modulation and new error correction schemes that are complementary to rank modulation. We layer error correction on top of the drift tolerant rank modulation without degrading the performance of either process. We provide an optimal failure encoding by only adding cells for failed cells, without incurring additional storage overhead. Here we describe the process of converting an integer number to a string of symbols for handling resistance drift in MLC PCM. We also modify the strings to handle 1, 2, and 3 stuck-at cells, where the cell failure can be due to probe delamination or probe degradation. We assume these failure modes map to the maximum and minimum states of the cell and that the failed cells can be differentiated from normal working cells. We describe an algorithm and provide an example that generates a string

of length $2m$ over an alphabet $S = \{0, \ldots, m-1\}$ of size m where each symbol occurs exactly twice. First, the number of such strings is given by

$$N = \prod_{j=0}^{m-2} \binom{2m-2j}{2}$$

Our algorithm takes an integer $i \in \{0, \ldots, N-1\}$ as input and produces the j^{th} string in this collection. The algorithm involves two parts:

- The first part is to write $i = \{r_0, \ldots, r_{m-2}\}$ where $r_j \in \{0, \ldots, \binom{2r-2j}{2} - 1\}$.
- In the next part, at step $j = 0$ we use r_0 to pick $S_0 \subset \{1, \ldots, 2m\}$ of size 2 and assign two positions the value $m-1$. This leaves us with $S \setminus S_0$ of size $2m-2$ positions to assign. Recursively at step $j \in \{1, \ldots, m-2\}$, we have $2m-2j$ positions left to be assigned, which are given by $S \setminus \cup_{t=0}^{j-1} S_t$. We use assign the index $r_j \in \binom{2m-2j}{2}$ to assign some two position S_j the value $m-j-1$. At the end, we are left with two positions $S \setminus \cup_{t=0}^{m-2} S_t$ which are assigned the value 0.

For the first part, we need to write $i \in [N]$ as $(r_0, \ldots, r_{m-2})$ and we are just performing the translation. To do this, we write

$$i \quad = q_1 \binom{2m}{2} + r_0, \quad r_0 \in \{0, \ldots, \binom{2m}{2} - 1\}$$

and for $j \in \{1, \ldots, m-2\}$,

$$q_j \quad = q_{j+1} \binom{2m-2j}{2} + r_j, \quad r_j \in \{0, \ldots, \binom{2m-2j}{2} - 1\}$$

Note that in the last step we have

$$q_{m-2} = r_{m-2}.$$

For our example, we assume an $[8,5,1]$ where $i = 1,001$, the equations are:

$$\begin{aligned} 1,001 &= q_1 \binom{8}{2} + r_0 \\ q_1 &= q_2 \binom{6}{2} + r_1 \\ q_2 &= q_3 \binom{6}{2} + r_2 \\ q_3 &= r_3 \end{aligned}$$

$$\begin{aligned} 1,001 &= 35 \times 28 + 21 \\ 35 &= 2 \times 15 + 5 \\ 2 &= 0 \times 6 + 2 \\ q_3 &= r_3 = 0 \end{aligned}$$

Given values for $r_0, \ldots, r_{m-2}$, it is easy to recover i by recomputing $q_{m-2}, q_{m-3}, \ldots, q_1$ and then i by the above equations. In *all* cases, $q_3 = r_3 = 0$.

For the second, we need a procedure which takes a set $S_{2k} = \{s_1, \ldots, s_{2k}\}$ and an index $i_k \in \{0, \ldots, \binom{2k}{2}\}$ and produces a two element subset of S_{2k}. This is done as follows:

Let $t, t' \in \{1, \ldots, 2k\}$ be such that

$$\begin{aligned} \binom{t-1}{2} < r_k + 1 \leq \binom{t}{2} \\ t' = r_k + 1 - \binom{t-1}{2} \end{aligned}$$

Output $\{s_{t'}, s_t\}$.

For our example, the coordinate space or indices are {1, 2, 3, 4, 5, 6, 7, 8}

$$\begin{aligned} \binom{t-1}{2} < 22 \leq \binom{t}{2} \\ \binom{7}{2} < 22 \leq \binom{8}{2} \\ t = 8 \\ t' = r_k + 1 - \binom{t-1}{2} \\ t' = 22 - 21 = 1 \end{aligned}$$

3's is placed in coordinates {8,1}. This leaves us with the six positions { 2, 3, 4, 5, 6, 7}.

$$\begin{aligned} \binom{t-1}{2} < 6 \leq \binom{t}{2} \\ \binom{3}{2} < 6 \leq \binom{4}{2} \\ t = 4 \\ t' = r_k + 1 - \binom{t-1}{2} \\ t' = 6 - 3 = 3 \end{aligned}$$

2's is placed in coordinates/indices {4,3} or positions labeled {5,4} from {2, 3, 4, 5, 6, 7,} which leaves { 2, 3, 6, 7}.

$$\binom{t-1}{2} < 3 \leq \binom{t}{2}$$
$$\binom{2}{2} < 3 \leq \binom{3}{2}$$
$$t = 3$$
$$t' = r_k + 1 - \binom{t-1}{2}$$
$$t' = 3 - 1 = 2$$

1's is placed in coordinates/indices {3,2} or positions labeled {6,3} from {2, 3, 6, 7,} leaving positions { 2, 7} for the 0's. The resulting string to write on to memory is {3, 0, 1, 2, 2, 1, 0, 3}.

For the decode step, we need to be able to invert this procedure efficiently. In other words, we are given a two element subset $(s_{t'}, s_t)$ of S_{2k} where $1 \leq t' < t \leq 2k$, and we want its index $r_k \in \{0, \ldots, \binom{2k}{2}\}$. The index is given by

$$r_k = \binom{t-1}{2} + t' - 1.$$

For our example, we reconstruct the sets starting with 00 and going up to 33. The coordination locations of 00 are fixed, so we move on to 11, with a string that looks like {0, 1, 1, 0}. The 1's are in location {3, 2}, providing t and $t-1$, which translates back to $r_2 = 2$.. Next, we add 2 back in and get the string {0, 1, 2, 2, 1, 0} for the coordinates {4,3}, which translates back to $r_1 = 5$. Adding the 3's back in provides the final coordinates {8,1}, which translates back to $r_0 = 21$. Knowing $q_3 = 0$, is the final piece of information required to reverse the encoding:

$$\begin{aligned} i &= q_1\tbinom{8}{2} + 21 \\ q_1 &= q_2\tbinom{6}{2} + 5 \\ q_2 &= q_3\tbinom{6}{2} + 2 \\ q_3 &= r_3 = 0 \end{aligned}$$

$$\begin{aligned} 1{,}001 &= 31\tbinom{8}{2} + 21 \\ 31 &= 2\tbinom{6}{2} + 5 \\ 2 &= 0\tbinom{6}{2} + 2 \\ q_3 &= r_3 = 0 \end{aligned}$$

4.1. 2-bit MLC with 1 stuck-at cell

The process is the same for 1-cell stuck-at errors. If we use 2-bit MLC as the example. We can specify the code like this $[8, 4, 2]$.

The message is a string of 4 symbols over an alpabet of size 4. We view this as a number $i \in \{0, \ldots, 4^4 - 1\}$. We use this to generate a string s' of length 7 which has exactly 2 3's, 2 2's, 2 1's and a single 0, using an algorithm similar to that used before. We would like to write the string $s = 0s'$ which has a 0 in position 1. Note that s contains every symbol in $\{0, \ldots, 3\}$ exactly twice.

But since there is one stuck-at fault, we may be unable to write s. Let $s + a$ denote the string $(s_1, +a, \ldots, s_8 + a)$ where we add a displacement a to each co-ordinate (and the addition is mod 4). Note that there exists some a such that we can write $s + a$ to memory, regardless of the location of the stuck-at fault.

For decoding, we are given the string $r = s + a$. We can recover the displacement a by reading position 1. By subtracting this from the string in memory, we can recover $s = 0s'$. From s', we can recover the message i using a similar algorithm as before.

In this scenario, we are using 7 symbols to encode a number of size 2^8. We use the last 7 symbols and add an anchor symbol, say 0, to the beginning of the 7 symbols to get a codeword size of 8. Once we know the error location and value at that location, we calculate the difference between the cell we want to write and the stuck-at location. We add this difference to all the cells mod 2^m (or 4). On decode, we can take the value of the first cell and add it to the 7 other cells, one-by-one and mod 2^m (or 4) to recover the original 7 symbol string.

For example, suppose we had the string ({ anchor , string}) { 0 , 1 0 2 3 1 2 3} and the error location is the left-most 2 and it is stuck in the low or 0 state, the string we write to memory is {2 , 3 2 0 1 3 0 1 }. Decode adds 2 then mod 4 to each cell to recover the original string.

Building the string from the number happens in the same way, but with the restriction that the pair of 0's must be partially determined. One 0 is always placed at index/coordinate 1. We must solve the above equations for the rest:

$$\begin{aligned} i &= q_1\tbinom{7}{2} + r_0 \\ q_1 &= q_2\tbinom{5}{2} + r_1 \\ q_2 &= q_3\tbinom{3}{2} + r_2 \\ q_3 &= r_3 = 0 \end{aligned}$$

The encode and decode of the string is done the same way, just the number of sets is reduced because we grab one symbol as the anchor.

4.2. Correcting 2 or 3 errors

These schemes have similar overall structure. Let $k \in \{2,3\}$ be the number of errors we wish to correct. We set aside k anchor symbols $M = \{m_1, \ldots, m_k\} \subset \{1, \ldots, m-2\}$. Note that anchors are distinct from 0 and $m-1$. Our encoding will first map messages to strings of length $n-k$, which contain (roughly) equal number of symbols from $\bar{M}$. In particular, 0 and $m-1$ will occur at least k times in these strings.

Fix one such string s'. We will try to write the string $s = m_1, \ldots, m_k, s'$ to memory, which contains the k anchors in the first k positions followed by s'. But the presence of stuck-at faults implies that some indices are i_1, i_2, i_3 are stuck (either at 0 or at $m-1$). So we may not be able to write s'.

To overcome this, we permute indices of the string using a family of (k-wise independent) permutations Π_k on $[n]$. Π_k has the strong property that for any k target indices $i_1, \ldots, i_k$ and any k indices $i'_1, \ldots, i'_k$, there exists a unique permutation $\pi \in \Pi_k$ such that $\pi(i'_j) = i_j$ for every $i \in \{1, \ldots, k\}$. These families only exist for certain values of n: n must be a prime power for $k = 2$ and a prime power $+1$ for $k = 3$. Such families of permutations are not known to exist for $k \geq 4$.

Returning to our problem, assume that $i_1, \ldots, i_k$ are stuck (some to 0, some to $m-1$). We will find indices $i'_1, \ldots, i'_k$ which are currently mapped by s' to to these values (0 or $m-1$). Note that there could be several possible choices for $i'_1, \ldots, i'_k$, in which case we pick one arbitrarily. We find the unique permutation π such that $\pi(i'_j) = i_j$. We then write the codeword $\pi(s)$ to memory. The reason this works is because if i_j is currently stuck at 0, π maps the location i'_j (which currently holds the value 0 in s) to it.

This completes the description of the encoding, we now turn to decoding. This is where the anchor symbols are used. Note that the codeword $\pi(s)$ contains a unique occurrence of $m_1, \ldots, m_k$. We know that in s, these symbols occur in locations $1, \ldots, k$. This fixes the permutation π uniquely, by the property of Π_k. By inverting the permutation, we recover s and s', and hence the original message.

More about the families Π_k:

- Assume that n is a prime power, so there exists a finite field $\mathbb{F}_n$. Let $\pi_{a,b}(x) = ax + b$ where $a \in \mathbb{F} \setminus \{0\}$ and $b \in \mathbb{F}$.
- Assume that n is a prime power + 1. Take the set of elements to be $\mathbb{F}_{n-1} \cup \infty$. Π_3 is given by the group of Mobius transformations $ax + b/(cx + d)$ with $ad - bc = 1$.

4.3. 4-bit MLC with 2 or 3 stuck-at cell

In the scenario where there are 2 or 3 stuck-at cells in a 4-bit MLC PCM, we use 2 or 3 anchor values, respectively. These strings differ from the previous construction in that the strings have a non-uniform distribution of symbols, with the anchor symbols appearing only once. The anchor values can be any value except the highest and lowest values. As an example, we will use the $[29, 20, 3]$ code.

For 2-bit stuck-at cells, the transformation of the number to a string is the same as described above. In this case, we map 2^{80} number on to 27 symbols and reserve the first two symbols in the string for the anchors. Let us assume the anchor values are 7 and 8 and we will map 15's to the two error locations. In converting the number to a string, we have the {position, symbol} mappings: {1, 7}, {2,8}, {4,15}, and {6,15}. Furthermore, cell positions 2 and 5 are stuck-at cells.

For 2 stuck-at cells, the equation $y = ax + b$ provide the linear shuffle, where the original positions of the values used to cover the faulty cells provide the x coordinates and the coordinates of the faulty cells provide the y coordinates. This equation maps all the combinations of new and old coordinates on the same line. Because we know the starting location of the anchor values and we read back where they moved to, we can reconstruct the linear equation and solve for x to move the cells back.

These operations happen over a finite field, guaranteeing a unique solution, where a and b are integer values. For our example, $x_1 = 4$ and $y_1 = 2$ and $x_2 = 6$ and $y_2 = 5$. Substituting these values into the equation gives us: $2 = 4a + b$ and $5 = 6a + b$. This simplifies to $2a = 3$. Because we operate over a finite field and the codeword is a prime power, we can use the identity properties to solve this equation, where all operations are *mod*29. The inverse value for 2 mod 29 is 15 (or 30 mod 29 = 1), so we multiple both sides by 15 and mod each side by 29: $2 \times 15 \times a \bmod 29 = 3 \times 15 \bmod 29$ resulting in $a = 3 \times 15 \bmod 29 = 16$ and $b = 2 - 64 \bmod 29 = 25$. $y = 16x + 25 \bmod 29$. In this scenario, the anchor values 7 and 8 map to coordinate positions 12 and 28, respectively. $12 = 16x + 25 \bmod 29 \Rightarrow -13 = 16x \bmod 29$ or $16 = 16x, x = 1$. Likewise, $28 = 16x + 25 \bmod 29 \Rightarrow 3 = 16x \bmod 29$ or $32 = 16x, x = 2$.

For 3 stuck-at cells, we use the following equation to shuffle the symbols in the string around, moving the +/- ∞ symbols to cover the stuck-at cells positions : $y = \frac{ax+b}{cx+d}$, where $ad - bc = 1$, and ∞ maps to the highest or lowest symbol. In this case, there are three unique anchor values that enable constructing the equation and unshuffling the string.

5. Mitigating Drift Induced Soft Errors

Rank modulation addresses the resistance drift common in MLC PCM for a bounded amount of time. Over time, the resistance values of cells could converge or cross over, introducing soft errors into the data. [18] demonstrated this to be a concern when the cell resistance values where uniformly distributed in log space, with fixed guard bands and large resistance distribution around the mean. Under these assumptions, drift-induced soft errors can occur in a reasonably large memory array on the order of a few seconds. For the statical analysis, [17] provides a good foundation. Resistance drift over time is quantified by the equation below. This relationship of time and resistance is linear in log-log space and as [6] demonstrates, is not influenced by an electrical field increasing the drift over time, ruling out an major contribution by tunneling. Table 1 provides the data for the states, guard bands, and deviations (in log base 10) common in literature used in [17, 18] for their analysis. Equation 1 provides the drift resistance equation. Note from the table that we are using 3 σ as the guard band and it is assumed that programming the initial value in the cell is within 2.75 σ. We can also rule out the lower half of the symbols and focus on only on the upper half of combinations based on the drift coefficients. This reduces the number of candidate cells to analyze.

$$R(t) = R_0(\frac{t}{t_0})^v$$

		$log_{10}R$		v	
cell level	data	mean	dev	mean	SDMR
1	00	3.0		0.001	
2	01	4.0	0.17	0.02	40%
3	11	5.0		0.06	
4	10	6.0		0.1	

Table 1: Resistance and drift values for 4-level cell PCM, where SDMR= standard deviatio to mean ratio.

Rank modulation removes the fixed guard bands that have been used in previous studies [17, 18] because the cells drift together and the decoding is based on relative ordering of the cells and not the absolute value. This means that a soft error occurs when the two different neighboring cell values are equal or cross over. If they are equal, they can't be differentiated and if they cross over, the cell values are exchanged. Rank modulation does not completely solve the drift problem, but does provide a longer time interval before the error will occur, making scrubbing/wear-leveling of old data still necessary.

However, by setting PCM cell resistance values to non-uniform values in the resistance log space [10] and tightening the resistance distributions (longer programming time) we can significantly increase the time before drift-induced soft errors occur, all but eliminating the refresh process that would significantly reduce the lifetime of wear-out prone non-volatile memories. [17, 18] assumes a resistance value distribution of 2.75 σ, which is very close to the fixed guard band. Reducing this resistance distribution to 1.375 σ requires more time to program the cell, but also provides more resistance space for the cells to drift. Table 2 shows a non-uniform distribution of the levels in the resistance space that mirrors a similar resistance distribution shown in later work [10], which clusters levels 1-3 much closer than we have in the table. Furthermore, we can reduce the resistance deviation and dramatically reduce the time before a drift-induced software error occurs. For the values in Table 2, the time before a soft error is approximately 5 days. Many more combinations of feasible cell data levels and associated deviation metrics could yield even longer data lifetimes.

		$log_{10}R$		v	
cell level	data	mean	dev	mean	SDMR
1	00	3.0		0.001	
2	01	4.0	0.17	0.02	40%
3	11	5.2		0.06	
4	10	7.0		0.1	

Table 2: Non-uniform resistance distribution and previous drift values for 4-level cell PCM, where SDMR= standard deviatio to mean ratio.

Finally, we could use redundant cells in the spare block to store additional information about the location of the top rank values or other information to correct for soft errors. Table 3 shows which encodings provide additional cells that can be used for this purpose.

Overall, we can incur longer latency programming times or add more information in order to address the issue of drift-induced soft errors. In our PCM evaluation, we assumes 32GB, 4-channel, 8-bank, DRAM with 12.8GB/s, full bandwidth with 2 simultaneous write on averags. In this case, the memory system lasts 3-5 years, assuming wear-level. The interval between writes to the same memory location is about 5 seconds, well below the 5 days required by drift and non-uniform cell resistance levels. Thus, even if we have to move data in the system due to drift, the system will still survive longer than the

code	Stuck-at	cells/block	cells/encoding	spare size	spare sub-block
2-bit MLC					
[20, 16, 1]	0	320	320	–	0
[12, 8, 2]	1	320	384	64	1/4 (80)
[8, 4, 2]	1	320	512	192	1 (320)]
4-bit MLC					
[64, 55, 1]	0	192	192	–	0
[48, 39, 2]	1	192	192	0	0
[41, 30, 3]	2	192	205	13	1/8 (24)
[28, 17, 4]	3	192	224	32	1/4 (48)

Table 3: Block size and spare cells required for each MLC encoding. Note: We start with a block size that can store the rank modulated 512 bits.

expected lifetime when we are not writing all the memory all the time.

6. Conclusions

This paper provides supplemental information for Zombie, a framework that can be used with prior and new error correction mechanisms to significantly improve SLC and MLC PCM lifetimes. Zombie uses memory that has been disabled due to exhaustion of intrinsic block error correction resources to keep memory that is still in service alive longer. Three of these mechanisms — ZombieXOR, ZombieECP, and ZombieERC— show endurance superior to various state-of-the-art SLC PCM error correction mechanisms. These mechanisms can also be used to correct stuck-at failures in MLC PCM, but doing so would require an additional compatible mechanism to tolerate drift. The fourth mechanism, ZombieMLC, is to our knowledge the first proposal to tolerate both stuck-at failures and drift in an integrated and seamless manner. ZombieMLC increases the lifetime of MLC PCM by over an order of magnitude compared to a standard rank-modulation mechanism, which tolerates only drift.

With all new technologies comes some uncertainty and we try to address these main issues in this paper. We provide the details that describe the fundamental behavior of proposed system. We provide the algorithms for reads and writes because they vary slightly depending on the Zombie scheme used. The analytical model provides a framework for analyzing new error correcting schemes for non-volatile memories that experience permanent cell failures. We were pleasantly surprised by how well the analytical model matched the simulation results. We then provide comprehensive details and examples for the ZombieMLC scheme. This is followed by insuring that MLC PCM is viable in the face of drift-induced soft errors. This is done by removing the fixed guard bands, using non-uniform initial resistance values in log space, and tighten the resistance value distribution when the cells are written.

In summary, the Zombie framework enriches the toolbox of designers seeking error correction mechanisms that match their specific system design goals.

References

[1] S. Ahn *et al.*, "Highly manufacturable high density phase change memory of 64mb and beyond," in *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, Dec. 2004, pp. 907 – 910.

[2] J. Aronson, A. Frieze, and B. G. Pittel, "Maximum matchings in sparse random graphs: Karp-sipser re-visited," 1997, pp. 111–178.

[3] G. Atwood, "The evolution of phase change memory," Micron, Tech. Rep., 2010.

[4] R. Azevedo, J. D. Davis, K. Strauss, P. Gopalan, M. Manasse, and S. Yekhanin, "Zombiememory: Extending memory lifetime by reviving dead blocks," in *ISCA*, 2013.

[5] Y. Hwang *et al.*, "Full integration and reliability evaluation of phase-change ram based on 0.24 mu;m-cmos technologies," in *2003 Symposium on VLSI Technology*, Jun. 2003.

[6] D. Ielmini, S. Lavizzari, D. Sharma, and A. Lacaita, "Physical interpretation, modeling and impact on phase change memory (pcm) reliability of resistance drift due to chalcogenide structural relaxation," in *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, 2007.

[7] E. Ipek *et al.*, "Dynamically replicated memory: building reliable systems from nanoscale resistive memories," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010.

[8] ITRS, "Emerging research devices," International Technology Roadmap for Semiconductors, Tech. Rep., 2009.

[9] R. M. Karp and M. Sipser, "Maximum matchings in sparse random graphs," in *IEEE Symposium on Foundations of Computer Science*, 1981.

[10] N. Papandreou, H. Pozidis, T. Mittelholzer, G. F. Close, M. Breitwisch, C. Lam, and E. Eleftheriou, "Drift-tolerant multilevel phase-change memory," in *Proceedings of the 3rd IEEE International Memory Workshop*, May 2011, pp. 1 – 4.

[11] M. K. Qureshi, "Pay-as-you-go: Low overhead hard-error correction for phase change memories," in *Proceedings of the 44th International Symposium on Microarchitecture*, 2011.

[12] D. Ralph and M. Stiles, "Spin transfer torques," *Journal of Magnetism and Magnetic Materials*, vol. 320, no. 7, pp. 1190 – 1216, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304885307010116

[13] S. Raoux *et al.*, "Phase-change random access memory: a scalable technology," *IBM Journal of Research and Development*, vol. 52, pp. 465–479, Jul. 2008.

[14] S. Schechter *et al.*, "Use ecp, not ecc, for hard failures in resistive memories," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, Jun. 2010.

[15] N. H. Seong *et al.*, "Safer: Stuck-at-fault error recovery for memories," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2010.

[16] D. B. Strukov *et al.*, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, 2008.

[17] W. Xu and T. Zhang, "Using time-aware memory sensing to address resistance drift issue in multi-level phase change memory," in *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, 2010, pp. 356–361.

[18] S. Yeo, N. H. Seong, and H.-H. S. Lee, "Can multi-level cell pcm be reliable and usable? analyzing the impact of resistance drift," in *Proceedings of the 10th Annual Workshop on Duplicating, Deconstructing and Debunking*, June 2012.

[19] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "Free-p: Protecting non-volatile memory against both hard and soft failures," in *Proceedings of the 17th Symposium on High Performance Computer Architecture*, 2011.