Journal of Circuits, Systems, and Computers
© World Scientific Publishing Company

CEPRAM: Compression for Endurance in PCM RAM

RODRIGO GONZALEZ-ALBERQUILLA[†], FERNANDO CASTRO[§], LUIS PINUEL* and FRANCISCO TIRADO+

ArTeCS Group, University Complutense of Madrid,
Madrid, 28040, Spain
[†]*rgalberquilla@ucm.es*
[§]*fcastror@ucm.es*
**lpinuel@ucm.es*
+ptirado@ucm.es

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

We deal with the endurance problem of Phase Change Memories (PCM) by proposing *Compression for Endurance in PCM RAM (CEPRAM)*, a technique to elongate the lifespan of PCM-based main memory through compression. We introduce a total of three compression schemes based on already existent schemes, but targeting compression for PCM based systems. We do a two-level evaluation. First, we quantify the performance of the compression, in terms of compressed size, bit-flips and how they are affected by errors. Next, we simulate these parameters in a statistical simulator to study how they affect the endurance of the system. Our simulation results reveal that our technique, which is built on top of *Error Correcting Pointers* (ECP) but using a high-performance cache-oriented compression algorithm modified to better suit our purpose, manages to further extend the lifetime of the memory system. In particular, it guarantees that at least half of the physical pages are in usable condition for 25% longer than *ECP*, which is slightly more than 5% more than a scheme that can correct 16 failures per block.

Keywords: PCM, endurance, compression, failures.

1. Introduction

Although DRAM has been the prevalent building block for main memories during many years, it has reached a limit in scalability and also faces problems of power due to the required periodical refresh of stored values. DRAM does not scale down well beyond 30nm [1]. This has driven researchers to look for new technologies. Some of them are Phase Change Memory RAM (PCM PRAM or just PCM), Spin Torque Transfer RAM (STT-RAM) [2] or Ferroelectric RAM (FRAM) [3]. While DRAM is a capacitive technology, meaning that logic values are stored as a charge in a condenser, these new technologies are resistive or magnetic technologies. In these emerging technologies, cells are made of a material that can be physically altered, changing its electrical impedance, making it possible to store logical values as different values of impedance.

PRAM is the candidate receiving the most attention, being it because it is compatible with CMOS process, because it can be scaled down beyond 16nm, or because –unlike DRAM– it does not require a periodical refresh. However, the endurance problem restricts the adoption of PCM as main memory for the next computer generation. The endurance is related with the amount of writes that a cell is likely to sustain before it fails, and in PCM technology this number is significantly lower than in DRAM. After a cell fails, it is not possible to change its value anymore and consequently the corresponding block and even the whole page it belongs to must be discarded. For these reasons, a lot of effort is being currently invested in making a reliable and durable memory system based on this emerging memory technology. As such, many researchers have tackled the endurance problem from a variety of aspects [4,5,6,7,8,9,10,11,12,13]. For example, doing wear leveling [10,14], to avoid early failures in hot-spots of the memory, or building a hybrid hierarchy [15,16,17] placing a DRAM based last-level cache over a PRAM based main memory. This paper approaches the problem from the perspective of compression to both reduce the amount of data written to the memory, thus reducing the wear, and to give a mean to encode information about errors in a memory block. In summary, our main concern is making memory blocks usable the longest possible. This requires detecting and surviving as many failures as we are able to.

The rest of the paper is arranged as follows: Section 2 provides some background. Section 3 presents our technique in detail. In Section 4 we show the simulation methodology and environment, and next, in Section 5 we evaluate our proposal. The state of the art is discussed in Section 6 and finally, Section 7 concludes.

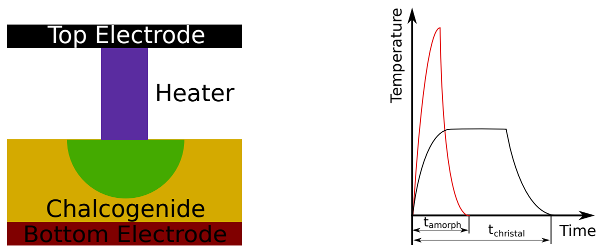
2. Motivation and Background

This section first introduces the concepts of *Phase Change Memory* and coding theory, doing special emphasis in compression. Then we introduce the concept of entropy and the ECP scheme, both key in the motivation of our technique.

2.1. *Phase Change Memory Technology*

Phase Changing Memory (PCM) is a memory technology that uses the electrical properties of a material to store memory [18]. More precisely, it uses the change in the electrical resistance of materials when they are transformed between an amorphous state and a crystalline state. The idea is not new, it first appeared in the 1960s, but it has been on the last years, with the use of chalcogenous alloys as $Ge_2Sb_2Te_5$ (or GST for short), when it has gained popularity.

Figure 1(a) depicts a PRAM cell, which consists of the two electrodes enclosing a heating element, and the chalcogenous material. The heater element is just a material that produces Joule heat when a current is driven through, warming the chalcogenous material. In order to write a logical value in the cell, the heating element is employed to apply electrical pulses to the chalcogenide, which changes the properties of the material resulting in two different physical states: amorphous (high



(a) (b)

Fig. 1. (a) PCM cell. (b) Heat pulses used to set and reset a PCM cell.

electrical resistivity) and crystalline (low resistance). Notably, if a high-intensity current pulse is applied, the material reaches over 600^oC and melts. Then, it is cooled down quickly, making it amorphous (RESET process). If the pulse is longer and with lower intensity, the material goes through an annealing process allowing the molecules to re-crystallize, lowering the electrical resistance (SET process). Thus, the chalcogenide switches easily, rapidly and in a reliable way between both states. Figure 1(b) shows graphically the heat pulses used to set and reset a PCM cell. The process for reading the stored value consists in applying a small current to the cell to measure its resistance.

The limitations of PCM as a replacement for DRAM are the higher write latency, and the limited write endurance. Next-generation PCM devices can endure just 10^7-10^9 writing cycles [19] vs 10^{15} in DRAM. The continuous expansions/contractions of the cell produced by write operations result in a detachment of the heater from the cell, leaving the cell in a *stuck-at* failure state. From that moment on the cell is still readable, but the value cannot be changed anymore.

On the other hand, PCM has some features that DRAM lacks. One of these good features is the fact that there are intermediate states between amorphous and crystalline. These states can be differentiated, allowing for multi-bit cells [20]. Another feature is that, not being a charge based technology as DRAM, PCM is not affected to particle-induced errors [21,22].

2.2. *Coding theory*

Error-correcting codes have been used to overcome errors in computer systems for a number of years. Any code $\mathcal{C}$ has 4 defining aspects:

- The **alphabet**: $\mathcal{A}$ is the set of symbols that form words of the code.
- The **length**: n is the number of symbols from $\mathcal{A}$ that form the codewords, this is, $\mathcal{C} \subset \mathcal{A}^n$.
- The **size**: k is the amount of different codewords in the code: $k = |\mathcal{C}|$.
- The **distance**: d is the minimum of symbols that differ between any two words of the code. The number of errors that $\mathcal{C}$ can correct is $t = (d-1)/2$.

Compression is an application of coding theory that tries to transform symbols, or collections of them, into symbols of a (maybe different) output alphabet, such that the frequent cases use the least amount of symbols as possible. Common techniques used are ordering symbols by frequency and encode them as new strings such that the common ones require less symbols than the uncommon ones (Huffman encoding) or storing in a dictionary symbols/strings as they are processed to encode subsequent occurrences as a backward reference. This achieves a reduction in the average amount of symbols required to store/transmit information.

There have been some proposals about using compression at different points in the memory hierarchy [23,24,25], but they had in mind speeding up bus transactions or virtually expanding the available space. In this proposal, we have in mind using the healthy bits of a block to store the result of compressing the data in the block. This requires the data to be compressible in, at most, as many bits as we have remaining healthy in the block. The entropy as a measure of the information conveyed by a word gives some insight of how well data will compress, regardless of the compression scheme used.

2.3. *Entropy as a measure of compressibility*

In information theory, entropy is a way to estimate how much information is conveyed by a symbol, a word or a whole text. Entropy is related with the probability of the symbols. For example, if we have $\mathcal{A} = \{0, 1\}$, and we consider the word "00000010000100000010″, the probability of a symbol in the word to be 0 is 17/20 and the probability of 1 is 3/20. In this sense, the statement *the n^{th} symbol is a 1* conveys much more information than the statement *the n^{th} symbol is a 0*, because 0 is the most likely value. The existence of this imbalance lowers the entropy, and we could just store the positions of the 1s to compress the word. On the other hand, if we take a look at the word "01100111001100110010″, P(0)=P(1)=10/20=1/2. This makes the symbols totally unpredictable, and thus the word is not compressible.

The formula to calculate the entropy of a language/word, L, from a $q-$ary alphabet, this is $|\mathcal{A}| = q$, is:

$$E(L) = -\sum_{s\in\mathcal{A}} (p_s \cdot \log(p_s))$$

where p_s is the probability of s appearing in L. If the base for log is 2, the unit for $E(L)$ is $bits/symbol$.

In the following we present two different values for the entropy of applications from the suite SPEC CPU2006. We consider byte symbols: $\mathcal{A} = \{0x00, 0x01, ...0xff\}$, and the words are cache blocks, that in our target architecture are 64-byte.

- Average Entropy: is the entropy of all blocks that are evicted from LLC and written back to main memory averaged. It hints about the regularity

Table 1. Meaning of the different possible values of Total and Average entropy.

Average	Total	Meaning
low	low	Both the symbols and the language are regular ⇒ Highly compressible.
low	high	Although the language as itself is not regular, words are, and word-level compression works fine.
high	low	Words are not compressible, but the language is ⇒ It is possible to encode symbols in a different way, so words are still 64 symbols but each symbol smaller in size.
high	high	Compression is unlikely to work well.

of the symbols in a word or the absence thereof. Small values mean that a few symbols are repeated in the same word, and there are a few symbols that only appear a small number of times. In contrast, big values mean that there is not predictability in the symbols of a word, because most of the symbols differ from one another. Given that words are 64 symbols long, the entropy ranges from 0 to $6 = \log_2(64)$.

- Total Entropy: is the entropy of the language formed by all blocks (the multiplicity is the probability of each word) evicted from LLC and written back to main memory. It gives information about the whole data footprint. Small values means that words are formed, mainly, by a small set of symbols that appear in most of the words, although it does not tell us anything about how many times these symbols appears inside a word, to that end we have the Average Entropy. A big value means that the footprint is not regular. Since $|A| = 256$, this value ranges from 0 to $8 = \log_2 256$.

Table 1 shows the meaning of Average and Total entropy when considered together. The best case is to have a low Average entropy, because that means compression goes well, and we can stick to it. If Average entropy is high but Total entropy is low, programs would need to go through a *common symbol extraction* phase, then a function $f : \mathcal{A} \to \cup_{n=0}^{N}\{0,1\}^n$ such as Huffman encoding built to re-encode the alphabet to achieve symbol-level compression. Although it is possible, the hardware resources to support this are larger.

Figure 2 holds the values for Average, Max and Total entropy for the applications from the suite SPEC CPU2006 with a 2MB LLC. Max entropy refers to the maximum entropy found among the evicted words. If it is 6 means that at least one block in which all the bytes were different is evicted from the LLC. This number gives a bound for the worst case, and it is interesting the fact that for a big amount of benchmarks this number is lower than 5, meaning that there is more than 16.6% redundancy in each word. Concerning Average Entropy, the results are promising, even for floating point applications, that have much more variability than integer applications. For all applications but one, the Average Entropy is smaller than 2.5, what hints for good compression rates.

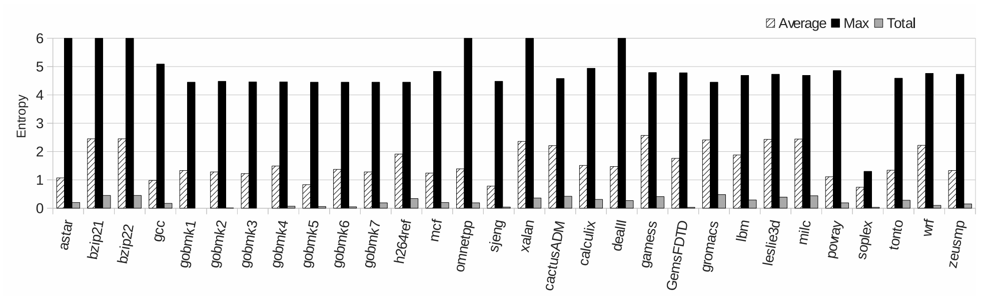


Fig. 2. Average, Max entropy per block, and Total language entropy for a 2MB LLC.

Concerning Total Entropy, the values are really low, this is due to the large amount of zeros that are written to memory. These values vary slightly with the cache size (we also explored 1MB and 4MB LLC sizes). This is because as the LLC size grows, the amount of write-backs decreases and the statistical variety with it. The lack of enough statistical variety may bias the value making it differ from the value experienced when different processes share the cache and the effective size used by each process is smaller than the actual size.

2.4. *Error Correcting Pointers (ECP)*

The usage of ECP to survive stuck-at faults has been one of the most successful techniques in the last few years. They were introduced y Schechter et. in [26], outperforming all techniques proposed to the date by large.

The idea behind ECP is using pairs $< p_i, r_i >$ of pointers (p_i) and replacement cells (r_i), such that when a cell in the block fails, a pair is allocated. The pointer stores the index of the cell to mark the failure and the replacement cell is used to store the value that would be held in that cell. With an overhead lower than 12.5%, a 512-bit block can be provided with 6 pairs. There are also 6 extra bits marking whether or not a given pair has been allocated .

When a block is written to, the actually-written value is compared with the intended value. If a new discrepancy is discovered, a pair $< p_i, r_i >$ is allocated, and the index of the cell in the block is written to p_i. All replacement cells are also updated with the new value. If any of them fails, let's say r_j, a new pair $< p_i, r_i >$ is allocated. p_i is set to p_j to point the same cell, and r_j takes the intended value.

When a block is read, all the allocated pointers are read and the failing cells values are substituted by the values in the replacement cells. This is done in index order, so if $i > j$ and $p_i = p_j$, the value used is r_i. This scenario happens if and only if r_i has experienced a failure. This priority-based substitution allows to correct up to 6 failures in both the block and the extra storage.

In the next section we show the foundations of a technique using compression which is used to extend the lifetime of a device beyond the limits of ECP.

3. Technique

In this section we present three techniques: 1) COMP, that compresses the information associated to a block and fit it in the healthy bits of the block to expand the effective lifetime, 2) $COMP_{CP}$, which is a variation of COMP using a compressing scheme much better suited for the context and 3) CEPRAM, which uses fine grain block pairing plus backspace capabilities to expand the effective lifetime even further.

In the following, we introduce the compression algorithms from the initial definition, explaining the modifications we have made along with COMP and $COMP_{CP}$. Next, we show how CEPRAM works using block-level pairing. After that, we remind how wear leveling techniques work, and how they behave when using our techniques. This section is concluded by a rough estimate of the lifetime of a system using CEPRAM.

3.1. *COMP*

3.1.1. *No-Table LZW compression (NTZip)*

LZW is a traditional text compression algorithm [27] used in many data formats and applications. We take LZW as the starting point for our compression scheme. LZW considers a dictionary preloaded with the symbols of the alphabet, in this case $\mathcal{A} = \{00_{hex}, 01_{hex}, ..., ff_{hex}\}$. Then symbols from the input are concatenated until the word they form is not in the dictionary. Then, the index of the prefix is output with the width necessary to write the size of the dictionary, $\lceil \log_2(sizeof(Dic)) \rceil$, the whole word added to the end of the dictionary, and the considered word becomes the last read symbol. This process is iterated until all the input is consumed.

LZW focuses on achieving good compression rates for the average cases, and does not care about the worst case, because there is enough statistical variety to absorb its effect resulting in a good compression ratio. In our context, we are dominated by worst case, therefore we need some extra mechanisms to further compress.

We propose the NTZip scheme, a variation of LZW in which the dictionary is not pre-loaded. It starts empty, and symbols are added as they appear. This requires output symbols to have one extra bit. The first bit of the output symbol has the following meaning:

- 0: The following 8 bits represent a symbol from $\mathcal{A}$ that was not present yet in the table.
- 1: The following $\lceil \log_2(sizeof(Dic)) \rceil$ bits represent a symbol from the table.

At first sight, this modification seems to penalize blocks with many different symbols, because it requires 9 bits per symbol: the leading 0, plus the 8 bits of the symbol itself. Giving a second thought, in LZW, after the first string is added to the dictionary, the width of output symbols is $\lceil \log_2(sizeof(Dic)) \rceil = 9$, as in NTZip, so we are not using more bits for the insertion of new symbols than LZW does. The main advantage of this modification is that $sizeof(Dic)$ starts at 0 and may grow up to 64, so output symbols that refer to the dictionary are, at most 7 bits long (1 preceding bit plus $log_2(64)$, as opposed to symbols in LZW that are, in our context, 9 bits long, except the first output symbol which is just 8 bits.

Algorithm 1 NTZip compression

```
$Dic \leftarrow \phi$
$S \leftarrow \lambda$                                   ▷ Output the first symbol, and insert it in Dic
$insert(concat(N), Dic))$
$output(concat(1, N))$                                   ▷ Proceed with the rest of the input
$N \leftarrow getChar()$
$ns \leftarrow true$
while $inputLeft()$ do
    while $concat(S, N) \in Dic$ do
        $S \leftarrow concat(S, N)$
        $N \leftarrow getChar()$
    end while
    if $ns$ then
        $ns \leftarrow false$
    else
        if $S$ is just a new symbol then
            $output(concat(1, S[0]))$
            $ns \leftarrow true$
        else
            $output(concat(0, index(S, Dic), width = \lceil \log_2(sizeof(Dic)))) \rceil$
        end if
    end if
    $insert(concat(S, N), Dic))$
    $S \leftarrow tail(concat(S, N)$
    $N \leftarrow \lambda$
end while
```

Algorithm 1 shows how the pseudo-code of the decoder looks after the modifications. First, we need to handle the first symbol separately, because $Dic = \phi$ is a special case. Another modification is the calls to output, that need to be prefixed by 0 or 1 depending on whether we found a new symbol that is not on the table yet or not. The last modification is the variable ns which controls that the same symbols are not output by error: when N gets a new symbol s, the condition $concat(S, N) \in Dic$ becomes false, we can assume S is formed of some previous symbols and $ns = false$, so the compressor outputs the index of S in Dic, inserts $concat(S, N)$ in the dictionary, sets $S \leftarrow s$ (given that $N = s \neq \lambda$, $tail(concat(S, N)) = s$) and N gets the empty symbol. The outermost loop starts a new iteration, $concat(S, N)$ cannot be in the dictionary, because $S = s$ which is a new symbol. Therefore, the inner loop is not entered, and the condition of the following *if* is satisfied. Then the new symbol is output, and inserted in the dictionary. The last two sentences of the outer loop leave S and N unchanged, because $N = \lambda$. In the next iteration $concat(S, N) \in Dic$, so N gets the next symbol from the input, and $concat(S, N) \notin Dic$. Here is when ns appears in the scene: if we did not check ns, the algorithm would just output the index of s in the dictionary, leading to s output twice, first as a new symbol and then as an indexed symbol, which is wrong. Having ns allows the algorithm to just insert $concat(S, N)$ in the dictionary and

Table 2. States for COMP and a proposed encoding to minimize bit-flips in transitions.

State	Encoding	Description
NF	0000	Initial state, no failures.
ECP_1	0001	First failure, corrected applying ECP.
ECP_2	0011	Second failure, corrected applying ECP.
ECP_3	0010	Third failure, corrected applying ECP.
ECP_4	0110	Fourth failure, corrected applying ECP.
ECP_5	1110	Fifth failure, corrected applying ECP.
ECP_6	1100	Sixth failure, corrected applying ECP.
COMP $_2$	1000	Seventh failure, discard bit pairs and apply compression.
COMP $_4$	1001	Eighth failure, discard bit fours and apply compression.
COMP $_{8a}$	1101	Ninth failure, discard whole bytes and apply compression.
COMP $_{8b}$	0101	Tenth failure, discard whole bytes and apply compression.

proceed. Using NTZip we achieve better results on average and almost the same worst case (just 1 more bit over a total of 575), as argued in Section 5.

3.1.2. *COMP operation*

Using NTZip requires hardware support to do the compression/decompression, but also requires extra bits to hold information about the block. In the design of COMP, we want to keep the overhead at bay: we target at most 12.5% space overhead, same as ECP_6 [26], SEC_{64} [28], $Pairing_8$ [29], $Wilkerson_4$ [30], and a perfect code correcting up to 9 errors. That makes for 64 bits of extra space.

The idea behind COMP, depicted in Figure 3, is using a ECP_6 scheme until the 6^{th} failure takes place (a). Once the 7^{th} failure arises (b), instead of discarding the block, it is compressed. Out of the 64 extra bits, 4 are used to encode that the block is compressed (left part of the picture), 56 bits hold 7 8-bit pointers that point to 7 bit pairs that are discarded (512 bits $\Rightarrow$ 256 pairs $\Rightarrow$ 8-bit pointers), so we can point to the 7 failing pairs, and the data is compressed as long as it fits in the remaining space, calculated by subtracting $7 * 2$ for the seven pair of bits discarded (failing bit plus accompanying bit due to granularity of addressing), from the block size: $512 - 7 * 2 = 498$ bits. When the 8^{th} failure takes place (c), the 4 bits encode that the block is compressed and there are 8 pointers 7-bit wide to groups of 4 bits that are discarded, because at least one fails. This allows us to use $512 - 8 * 4 = 480$ bits to store the compressed data. The next step is widening the discarded chunk size again, narrowing the pointer size to 6 bits. This allows us to discard 9 bytes when the 9^{th} failure happens (d), and 10 (e), when another cell fails. This method can survive up to 10 failures, as long as the compressed data fits in $512 - 10 * 8 = 432$ bits. If at some point the data is compressed and it does not fit in the block, the block is deemed useless and discarded. This is feasible, because we devote 4 bits to encode the state, as Table 2 shows.

The results for applying this technique improve ECP, if only because it is build on top of it, but are not very good. The problem is that we are limited by worst case. This means that it does not matter if a block compresses down to, let us

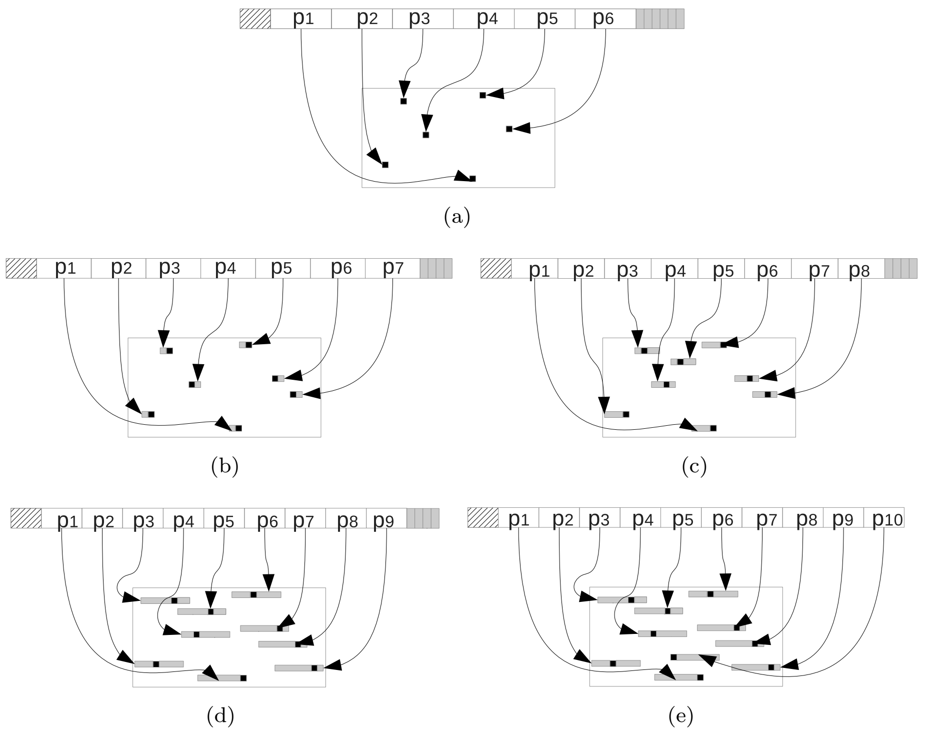


Fig. 3. COMP operation. The top horizontal bar symbolizes the 64 bits of meta-data, and the square models the data block. In the meta-data section color grey means unused bits. (a) ECP is used while the number of failures is 6 or less. There are up to 6 9-bit pointers pointing the failing bits and 6 replacement cells. (b) If the block has 7 failures, 7 8-bit pointers point to the failing pair. Out of each pair, only one bit is failing (black) and the other one is discarded, even if it is healthy. (c) There are 8 7-bit wide pointers to groups of 4 bits that are discarded because at least 1 bit is failing (black). (d) and (e) 9^{th} and 10^{th} errors occur: 9 and 10 6-bit pointers respectively to whole bytes that are discarded because at least one bit (black) is failing.

say, 300 bits, so it fits even with 10 failures and 10 bytes discarded, on average, because if in a write-back operation it does not fit in the available space, the block is deemed useless and discarded, and the whole page with it. Therefore, this is not the best context for compression. Another shortcoming of this scheme is discarding 10 bytes when there are 10 bits failing, the rate of wasted space is 10x. This motivates adapting NTZip to make it more suitable for our purpose.

3.1.3. *NTZip with backspace*

The main problem with COMP is that a lot of space is wasted due to the limitation in the size of the pointers. In this section we propose not doing explicit recording of the failing cells, given that stuck-at cells are still readable, and thus, the failure is exposed only 50% of the times. The rationale is that, if we assume the same probability of 0 and 1 in the values of each bit, the probability of a stuck-at bit being written a different value that the stuck-at value is 50%. This means that if we have n failing bits, on average only $\frac{n}{2}$ of the failures will manifest. Knowing this, we also propose NTZipBs, a modification of NTZip in which the symbol "0" of the output code encodes a backspace, that lets the code express that there have been a failure in the previous symbol, and should be fixed. This is done by modifying the symbol table (dictionary). The very first entry is allocated upon initialization. All match tests to index 0 return *false*, so the output symbol is never 0. If during the writing of a symbol a fault is exposed, then a 0-index symbol is output to mean that there is an error in the previous symbol. Algorithms 2 and 3 show the code for an optimistic simplification of NTZipBs that assumes that no failures are exposed in the backspace, and that no symbol is, due to failures, transformed into a backspace. The idea is the same as in NTZip, but when writing the block, if a failure is detected in one symbol, a backspace is inserted afterwards.

Algorithm 2 NTZipBs compression

```
$Dic \leftarrow \mathcal{A}$
$insert(< bs >, Dic)$
$S \leftarrow \lambda$
while $inputLeft()$ do
    $N \leftarrow getChar()$
    while $concat(S, N) \in Dic$ do
        $S \leftarrow concat(S, N)$
        $N \leftarrow getChar()$
    end while
    $output\_symbol(index(S, Dic), width = \lceil \log_2(sizeof(Dic)) \rceil$
    $insert(concat(S, N), Dic))$
    $S \leftarrow N$
end while
```

Algorithm 3 $output\_symbol(S : integer, width : integer)$

```
$error \leftarrow$ true
while $error$ do
    $write_memory(S, width)$
    $error \leftarrow check_memory(S, width)$
    if $error$ then
        $write_memory(0, width)$
    end if
end while
```

The two major shortcomings of this modification are:

- A block with n failures can require up to $n+1$ iterative writes: If in the first write we detect one failing cell, we perform a second write of just the tail of the block after the failure with the backspace and the rest of the symbols. If, in turn, this second write shows a failure, a third write is required, and so on up to $n + 1$ writes.
- The decoding of LZW, NTZip and NTZipBs is not parallelizable for such small blocks. The coding is not parallelizable either, but write-backs are

not in the critical path. This has a major impact, because it will insert non-negligible time overheads to LLC failures in the system “early”.

The first problem is not critical, it is reasonable to slow down writes as the system gets old. We assume that it is better to have some slow blocks than to discard some pages. The second problem is more important, and in Section 3.2 we address it. Prior to that, in the following we briefly analyze the hardware overhead and throughput penalty of the proposed schemes.

3.1.4. *Implementation overhead*

For the coding we require an automaton that writes, reads, and in case of a fault, regenerates the “queue” (from the faulting word onward) by inserting first a $< bs >$ (to remove the symbol where the fault is manifested). If the circuit is a comparator (read block == written block), it is needed to find the first mismatch, to perform a shift and to insert the $< bs >$. In the general case we require a stack (counter), since, if the error is in a $< bs >$, it is needed to insert the $< bs >$ again, and another $< bs >$ for the erroneous character in which the $< bs >$ become due to the fault. But obviously, these two new $< bs >$ are again error-sensitive, hence there is the need of a counter. The circuit iterates $O(\mathrm{n}/2)$, where n is the number of faults in the line, because we only incur overhead in case there exist exposed faults. In any case, being a writing process, it is out of the critical path, and as all the data are located in memory, we can do a design that places this circuit on the memory side, thus not consuming system bus bandwidth.

For the decoding process, as it is performed symbol by symbol, we require $16+2n$ steps, where n is the number of $< bs >$ in the given word. The biggest challenge with NTZip and therefore, with NTZipBs, is implementing the decoder. This challenge comes from two facts:

- Dependencies among symbols: it is possible to have each symbol depending on the previous symbol.
- Variable amount of symbols: the amount of symbols needed to compress a block depends on the contents of the block.

This means that, in the general case, we need to either serialize the decoding or have a form of speculation on the dependency and recover mechanism. The former has simpler hardware at the cost of higher latency. For the latter, assuming we want to decode 2 symbols in parallel, we would require a predictor for the dependency, two possible paths for each outcome, along with the recovery mechanism. For n symbols in parallel, we need $O(n)$ predictors ($n-1$ to be precise), and $O(n^2)$ data-paths ($(n-1)^2$). On top of this requirement, by the second observation, we cannot do a pipelined implementation, unless we provision for the worst case. For NTZip, that means a block holding 64 different one-byte values, which corresponds to 64 symbols. Putting everything together, that means either 64 stages, each equipped with a dictionary that can hold up to $n-1$ words of length $min(n-1, 10)$. The bound on the length comes from the fact that, in order to generate a symbol encoding n bytes, we need a symbol encoding $n-1$ bytes. Therefore, to get to a symbol that encodes 11 bytes, we need a symbol that encodes 10, one that encodes 9 and so on. If we add the lengths, $1 + 2 + ... + 11 = 66$, we would go beyond the length of a block, so we are guaranteed to generate symbols that encode, at most, 10 bytes. This amounts to, roughly, $2Kwords \approx 10Kbytes$ with an extra 64 cycles latency in the worst case. The average amount of symbols per block in the studied applications is around 45, so we would have a latency increase of 45 cycles, assuming our circuit is capable of doing early exits. If we consider a non-pipelined implementation, that would require, on average, 45 cycles per cache block, this is, 64 bytes. This would limit the bandwidth to $\approx \frac{45}{64} * clock\_frequencybytes/s$. This may not be enough for configurations in which many cores share the memory controller, but as the decoding of different blocks is independent, we can just replicate the hardware.

Although the cost in hardware is affordable, increasing the latency of loads in 45 cycles on average is undesirable. This motivates the search for a compression scheme better suited for this use case.

3.2. *COMPcp*

3.2.1. *C-Pack and C-PackBs*

Chen proposed C-Pack [23], a cache compression scheme targeting high performance. $COMP_{CP}$ is a variation of COMP using C-Pack instead of NTZip as compression scheme. We slightly adapt the hardware decoder employed in [23] by adding some logic to discard the failing sets of bits, which can done in a few gate levels.

C-Pack input alphabet are 4-byte symbols. Each byte can be in one of the following 3 categories: Z if it is $0x00$, M if it matches the byte in the same position inside the symbol of a word in the dictionary, or X if the byte is neither Z nor M.

Using C-Pack also requires discarding faulty bits as well as NTZip, what led us to modify it by including a *backspace* character (C-PackBs scheme). Table 3 shows the modification of pattern encoding done to C-Pack (Table 1 in [23]). The pattern *ZZZX* is eliminated. We do that without hurting compression much by initializing the dictionary with the word **0**, *i.e.*, the word formed by 32 zeros, so *ZZZX* is a subcase of *MMMX*, also we can recognize *ZZXX* as *MMXX* with **0**. We substitute it with the backspace ($< bs >$) and also shorten it in one bit.

In our C-PackBs, the $< bs >$ is, strictly speaking, not a *backspace*, but a correction symbol, always followed by 5 bits, to make a 1-byte symbol. The longest symbol is 34-bit long, requiring less than 5 whole bytes. In the 5 bits following $< bs >$ each 1 means that a byte contains at least one error, and accordingly, after the bs-byte, are as many bytes as 1s in the 5 bit pattern, each to be xor-ed with the failing byte of the previous symbol. Figure 4 depicts an example.

In Figure 4 (a) there is an extract of a compressed block. The first two symbols

Table 3. Pattern encoding for C-PackBs

Code	Pattern	Output	Length(b)
00	ZZZZ	(00)	2
01	MMMM	(01)bbbb	6
1101	MMMX	(1101)bbbbB	16
1100	MMXX	(1100)bbbbBB	24
10	XXXX	(10)BBBB	34
111	$< bs >$	(111)bbbbbB..B	16..48

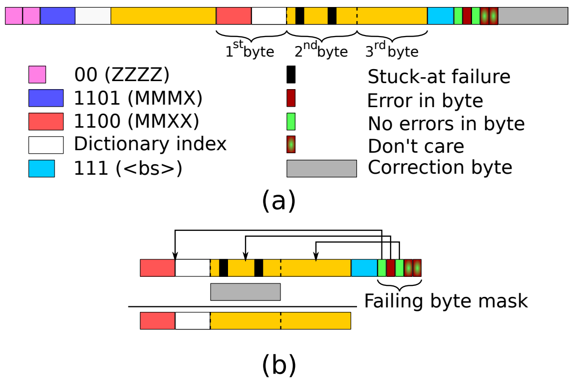


Fig. 4. Error handling in C-PackBs. (a) Compressed block. (b) Generation of correct symbol using the correction byte (in this case “01000100”).

correspond to the pattern *ZZZZ*. Next symbol is a *MMMX*, so following the symbol are 4 bits to index the dictionary, and the less significant byte. The fourth symbol is *MMXX*, which is the symbol containing failures, and therefore next symbol is $< bs >$. The failing symbol is comprised by the pattern descriptor, “1100” (in red), the 4-bit corresponding to the index in the dictionary of the matched word (in white), and the 2 least significant bytes of the uncompressed word (in yellow). The failing bits are in the 2nd byte (bits 6 and 2). After writing the symbol we discover the failures, so, instead of writing the next symbol, we insert a backspace (cyan), followed by 5 bits: 0, because the 1st byte contains no errors (green), 1 because the 2nd byte contains errors (deep red), and three more zeros. After it, a byte containing ones in the position of the failures is inserted. Then, the rest of the block is written. When we decode the word (Figure 4 (b)), the decoder detects the $< bs >$ symbol, and reads the following 5 bits. Given that there is only one bit set, it reads one correcting byte that is XOR-ed with the 2nd byte to produce the corrected symbol that can then be decoded. The two bytes ($< bs >$, 5-bit mask and correcting byte) are eliminated and the decoding proceeds. There are more complex situations in which the $< bs >$ contains errors or even situations in which the error is in the header of a symbol, resulting in an output symbol which length is different than expected. These situations only affect the step of writing into memory. The coding is unaffected, and the decoding is the same regardless of where the error happened, and consists in a pre-pass that detects $< bs >$ headers and corrects symbols.

3.2.2. *Implementation overhead*

The overhead of C-Pack [23] in silicon and latency is much more competitive than the one for NTZip. The authors report, for the studied technology nodes, a decode latency of 8 cycles, and an encode latency of 13 cycles. They also do a detailed description of the circuits and motivate the low overhead [23].

Concerning the versions with backspace, the cost is similar for both compression algorithms. One possible and simple implementation is a circuit that sequentially inspects each symbol of the read block and the following. In case of the latter not being a backspace, it shifts it, and iterates again over the next symbol and the one after it. In case of a $< bs >$, both symbols are discarded, the latest shifted symbol becomes the current symbol for the next iteration, and the symbol following the discarded pair becomes the next symbol. The need to *go back* upon detection of $< bs >$ and the possibility of having consecutive $< bs >$ is easily dealt with using a stack of up to 16 entries holding the position of the *current* symbol in the block, *i.e.*, 10 bits per entry. The control for this unit is very simple, as it has to check the second symbol for a $< bs >$ (shift + compare). And based on that, stack the position of the current symbol, or pop the top of the stack, until the stack is full (16 symbols) and the next symbol is not a $< bs >$. This means $16 + 2n$ cycles per block, where n is the amount of $< bs >$ in the block, with a negligible amount of hardware. Again, as the circuit is sequential, it may impose an undesired bound to bandwidth that can be easily address by replicating the proposed scheme.

3.3. ***CEPRAM***

3.3.1. *Further increasing the life through block-level pairing*

Our last proposal is CEPRAM: Memory compression with block-level pairing for an improved life-span in resistive memories. The idea of CEPRAM is working as $COMP_{CP}$ for the 10 first failures. CEPRAM keeps a pool of discarded pages. When a block hits the 11^{th} failure, if the pool is empty, the whole page is discarded from the system and added to the pool. If the pool is not empty, the first available block is allocated as "overflow" block of the failing one. This is done by using one of the available encodings for the state in the meta-data for this new state, *PairLeader*. Another one of those available encodings will be used for the linked block, *PairSlave*, and yet another one for when a block is not usable anymore *Useless*. The remaining 60 bits are used to store the physical address of the linked page. Actually, those 60 bits allow for writing the address with redundancy, to survive stuck-at failures in the meta-data region. The linked block has its state modified accordingly, and from then on, when the block is written to, the block is compressed with C-PackBs, using the second block to write data overflowing from the first, if at some point the compressed data does not fit in the 1024 bits of the combined blocks, then the *PairLeader* is discarded with its whole page, and added to the pool. In this initial proposal, the linked block is lost. The decoding of a *PairLeader* proceeds as usual, but continuing with the *PairSlave*, if after processing the first block the decoder has found less than 16 non $< bs >$-symbols.

3.3.2. *Wear Leveling*

In CEPRAM we use wear leveling techniques as in [31,15]. However, we use that only for the data part, the meta-data is not *wear-leveled.* The reason for this is that we need the 4 state bits to be reliable. If a line goes through all states, it will be around 3 writes to each bit. The probability of these bits to get *stuck-at* is really close to 0. Also we need the pointers to discarded blocks to be as healthy as possible, and the same for the last stage of a block life, when an overflow block has to be linked, because if a block cannot be linked or cannot transition states, the block (and the whole page with it) is immediately discarded. With this decision, we are trading more wear in the data bits for less wear in the more-critical meta-data bits.

When two blocks are paired, the leader suffers wear more often than the slave (overflow) block. Allowing for wear leveling inside the pair affects performance because the start of the block may be physically placed in the second block, and that scenario requires fetching the *PairLeader* from memory, getting the *PairSlave* address, fetching it, and then start the $< bs >$ expansion. If wear is leveled only inside each block, then the $< bs >$ expansion can be started as soon as the first block arrives, and the fetching of the second is done in background.

3.3.3. *Multiple Linking*

CEPRAM only links once, but an scheme can be devised in which we have plenty of overflow blocks. We discarded that option because that situation arises only when the system is close to malfunction, and there is no point in “prolonging the agony”.

3.3.4. *Lifetime estimation*

The average compressed block size is 320 bits. For every failure, in the worst case, we use 2 bytes, one for the $< bs >$ and the mask, and another one for the correction byte. If two failures are in the same symbol, then the algorithm may insert either 2 or 3 bytes, depending on the two failures being in the same byte or in different bytes. We can do an average case estimation of the lifetime measured in *survived failures.* If a block is 512 bits, on average, there are $512 - 320 = 192$ bits $= 24$ bytes, that let us correct up to $\left[\frac{24}{2}\right] = 12$ failures before discarding the first block. After pairing, the amount of available bits is $1024 - 320 = 704$ that allow for $\left[\frac{704}{16} = 44\right]$ failures corrected. On average, if the probability of a stuck-at failure to manifest is 0.5, this can correct on average 90 failures per paired block.

Table 4. Parameters of the memory hierarchy.

	DL1	**IL1**	**L2**	**L3**
Size	32KB	32KB	256KB	1,2 and 4MB
Associativity	8 ways	8 ways	8 ways	16 ways
Block size	64 bytes	64 bytes	64 bytes	64 bytes
Replacement policy	LRU	LRU	LRU	LRU

4. Simulation Environment

In this section we present the simulation environment and the methodology of simulation used to evaluate the technique proposed in the previous section. We start by introducing the applications used for benchmarking and following it is a description of the two simulation tools used.

Benchmarks: we use all the applications from the suite SPEC CPU2006 [32]. We have compiled the applications using GNU compilers, namely, *gcc*, *g++* and *gfortran* version 4.6. The optimization flags used are those enabled by the standard optimization flag *"-O2"*. All applications are run to completion with the test input.

Simulator for compression: for the compression schemes we use a memory hierarchy simulator based on the cache simulator from the SESC [33] processor simulator. This simulator is connected to a pintool [34] which instruments guest applications so the cache model takes the appropriate actions prior to every single memory access. Both instructions and data accesses are simulated. Table 4 gathers the parameters of the memory hierarchy simulated. Whenever the simulator evicts a block from the LLC, it performs the appropriate actions. For entropy and bit-flip probability calculations, these actions are just appropriately modifying the counters according to the contents of the block. For NTZipBs and C-PackBs, the actions are more complex. First, the block is compressed. Second, for $n = 1, 2, ..., 50$ a 512-bit wide error mask containing exactly n errors is generated. The compressed block is written into a single block using the error mask, and it is accounted as an overflow if the required size exceeds 512 bits. Otherwise, it is accounted for as a success. Next, for $n = 1, 2, ..., 50$ a 1024-bit wide error mask containing exactly n errors is generated. The compressed block is written into a single block using the error mask, and it is accounted as an overflow if the required size exceeds 1024 bits. Otherwise, it is accounted for as a success. This is done to differentiate the probability of overflow when the block is in its own from the probability when the block is paired. In addition, to calculate the amount of bit-flips, the "previous value" is kept, compressed and written for each mask, and both output are compared to count the number of bits that flip.

Memory system simulator: we use an in-house simulator. Doing faithful, cycle accurate simulation is hard and impractical, because it will require simulating workloads until the end of the lifetime of the system. For that reason, our simulator does a number of assumptions. First, we assume that there is an underlying wear-

Table 5. Memory system simulator parameters.

Page size	4KB
Row size	64 Bytes
Rank	1
Chips per rank	8
Bit lines per chip	$x8$
Lifetime distribution	$N(\mu = 10^8, \sigma = 2.0 \cdot 10^7)$
Pages	2000

leveling technique to evenly wear all the memory cells, or at least, those not devoted to actual data, depending on the technique. Next, we assume that memory chips store data in 512-bit blocks (rows), and that each contiguous block of memory is spread over eight chips. Finally, writes are performed at block level. When a block is written, each bit is modified with probability p extracted from the study on flip probability. This is needed, because compressing data will narrow the size of writes, but as a side effect will increase the probability of bits being flipped, because compression reduces size by eliminating regular patterns, therefore, compressed blocks are less regular and more prone to have more bits modified per write.

As shown in Table 5, we simulate a system with $4KB$ pages. Each technique is simulated by creating a number of memory pages. Each bit inside every page, including meta-data, is created with a lifetime randomly distributed according to a Gaussian distribution $N(\mu = 10^8, \sigma^2 = 2.0 \cdot 10^7)$. Initially, the wear ratio, w is calculated as a function of p and the number of pages in the system, as Equation 1 shows. Then, according to the wear ratio, lifetime calculation is performed, and the simulation proceeds by locating the next failing cell, wearing all the system accordingly, applying the actions corresponding to the simulated technique: allocating an ECP, discarding a page, pairing a page, ignoring it because an ECC is capable of correcting the error, *etc.* After actions are taken, w is updated if needed, and lifetime recalculated.

$$w = p \cdot \frac{\#starting\_pages}{\#alive\_pages} \tag{1}$$

w is affected by 2 factors:

(1) Page deaths: When physical pages die, the remaining pages need to "absorb" writes to that page. Due to the assumption of wear leveling techniques, that extra wear is evenly spread among the remaining pages. That requires recalculating w and the lifetime for all alive pages in the system.
(2) Technique application: When a technique takes an action that modifies either p or the size of block writes, the wear ratio of a block, a page or of the whole system may be modified. For example, when a block start being compressed, both p and the average compressed write size are modified requiring w to be recalculated along with the remaining lifetime.

5. Evaluation

In this section, we show the results of the evaluations of the technique proposed in Section 3. First, in Section 5.1 we show the performance of the compression schemes considered, in terms of frequency of best case, frequency of overflow and average compressed size. Next, in Section 5.2 we evaluate NTZipBs and C-PackBs in terms of survived failures, and evolution of the average bits flipped per write with the number of errors. To conclude the evaluation, in Section 5.3 we show the results of the simulation of a PCM based system in terms of the lifespan, comparing it to previous proposals.

5.1. *NTZip and C-Pack performance*

The two aspects of a compression scheme that we have chosen to focus on are :

- *% Constant Block*: it is the percentage of blocks that are constant, and therefore are the most compressible ones.
- *% Overflows*: it is the percentage of blocks that exceed 512 bits when compressed. It represents how often a block lies in the *worst case scenario*.

Figure 5 shows both aspects for NTZip and C-Pack (using SPEC CPU2006 applications and 2MB LLC. SPECINT, SPECFP and Aggregate refers to the weighted average considering integer, floating point and all applications respectively.

NTZip: With NTZip the constant blocks compress down to 48 bits, while the maximum compressed size is 576 bits: each byte is “new” so it is encoded as a 9-bit symbol, with a 1 identifying the new byte plus the byte itself.

The reason for the modification of LZW is that, for the case of the constant block, LZW manages only to compress down to 98 bits: first symbol is 8 bits, and the remaining 10 symbols required to encode the word ($\frac{(1+10)*10}{2} = 55 < 64 < 66 = \frac{(1+11)*11}{2}$) require 9 bits, because the number of symbols of the dictionary is greater than 256 after the first byte is encoded and output. For the smallest compression ratio, NTZip is 576 bits, while LZW only improves it by 1 bit: 1 symbol of 8 bits, plus 63 symbols 9-bit wide total 575 bits. It is easy to proof that for input sizes of 64 bytes, NTZip compresses always to smaller sizes than LZW but for the case of 64 different symbols.

NTZip does an amazing job compressing blocks from SPECINT applications. The one for which it performs the worse is *bzip*2. Since *bzip*2 is a compression program in itself this is totally reasonable, because compressed data does not recompress well. Compressing SPECFP is harder. The compression algorithms considered are not FP oriented, and therefore, the higher variability at byte level intrinsic to floating point data is the reason why they perform worse than for integer code. Nevertheless, on average 75% of the blocks that FP applications write back to memory compress to sizes ≤ 512, and 20% of the total are the same byte repeated over and over again.

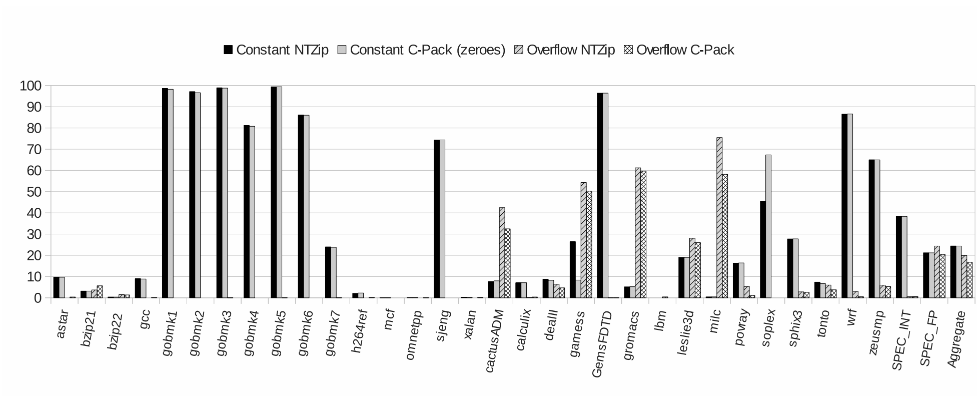


Fig. 5. Percentage of blocks evicted from LLC that are the same byte repeated 64 times (constant, zeros in C-Pack), and that compress with NTZip and C-Pack to more than 512 bits (overflow).

C-Pack: there is a slight difference with respect to NTZip. In NTZip, a block being the same byte repeated over and over again is the best case scenario. C-Pack is, in that respect, more restrictive. It requires the block to be all zeros, in which case C-Pack is able to compress it down to 32 bits, improving the 48 bits achieved by NTZip. Given that most of the constant blocks are zeros, this restriction is not a big constraint. Another difference in the numbers is that in the presence of a block with 64 different bytes, the compressed size for NTZip is 576 as discussed above, while C-Pack manages to use just 544 bits. Although Figure 5 is not enough to state that the compression ratio of C-Pack is higher than that of NTZip, it is a good reason to use C-Pack. Since CEPRAM pairs two blocks and writes the compressed block in the two blocks, having a smaller maximum size makes C-Pack very well suited for this purpose.

As shown in Figure 5, the numbers for C-Pack do not differ much from those achieved by NTZip. On the whole, C-Pack manages to keep overflows lower than NTZip, although the difference is not significant.

Regarding those blocks that neither are constant nor compress to more than 512 bits, Figure 6 shows for each application, the average compressed block size using NTZip and C-Pack. For these blocks, the distribution of the size to which they compress is pseudo-normal in both cases, differing from a normal more in the case of C-Pack. This is because being C-Pack a word-oriented algorithm, there are only a small number of different sizes that can be achieved.

Note that for NTZip, there is no general trend, not even if we distinguish between integer and floating point code. However, 335 bits is a low average size if we take into account that the input size is 512 bits, and with so few input symbols, just 64, it is hard to achieve high compression rates.

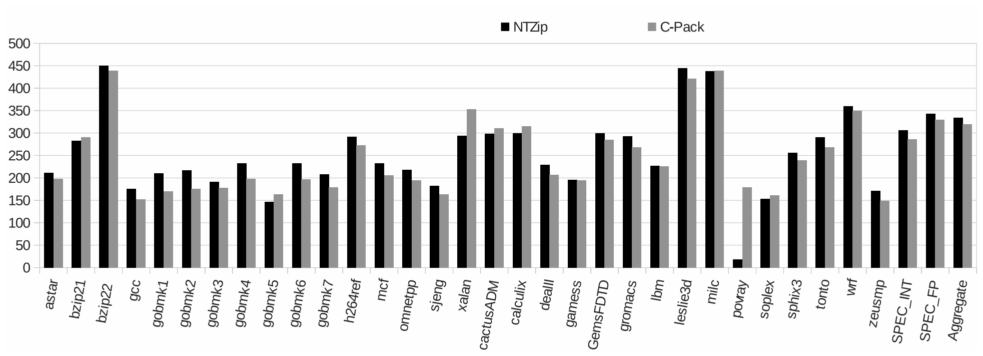


Fig. 6. Average compressed block size for both NTZip and C-Pack.

C-Pack as well as NTZip does not show any pattern or regularity in the average compressed size. There is no correlation between the average compressed size for both algorithms either. For some applications as $gobmk_{1,2,3,4}$, mcf or $zeusmp$ C-Pack outperforms NTZip, but for $xalancbmk$, $gobmk_5$ or $calculix$ it is NTZip outperforms C-Pack. There are also some applications ($bzip2$,$gamess$,$soplex$) for which the difference is negligible. Overall C-Pack achieves a smaller average size, but also with a higher coefficient of variance.

These results back up the design decision of changing the compression algorithm for C-Pack, because there is already a high performance encoder/decoder, especially designed for the place in the hierarchy we intend to place it.

5.2. *NTZipBs and C-PackBs performance*

When we augment NTZip with the backspace character the average compressed size is hard to quantify. Instead, we have taken the amount of (exposed) failures survived in a block pair as the figure of merit. This number, M, is the maximum number of exposed errors, N, for which after doing the following process for all blocks that are written back to memory, none reported a final size greater than 1024 bits (paired block):

(1) Calculate C as the result of applying NTZip to the B written-back block.
(2) Generate a random 1024-bit mask with N errors, $mask_e$.
(3) For each symbol $s \in C$
   - Write s and check $mask_e$.
   - If no errors are detected, proceed with next symbol.
   - If an error is detected, insert a backspace, write s and check for errors again.

The other feature we have focused on is the probability (ratio) of blocks that actually exceed 1024 bits when written back with $M+1$ errors. This second number

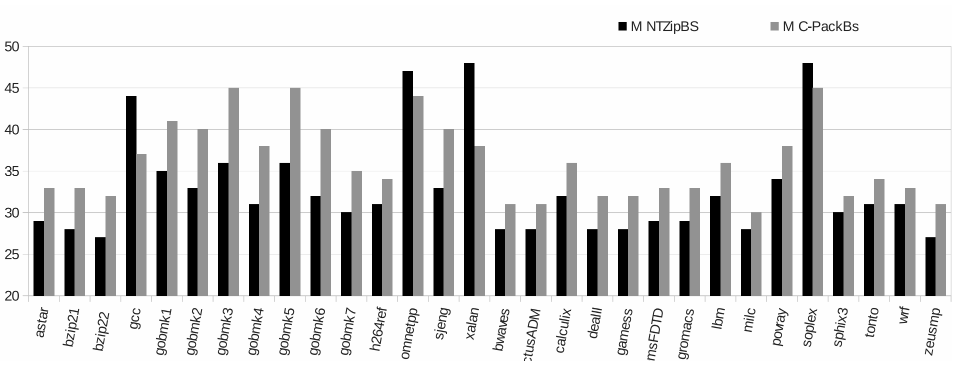


Fig. 7. Max. survived exposed failures (M) for NTZipBs and C-PackBs.

provides an insight on how long it will take for the block to fail after the $(M+1)^{th}$ failure is hit.

Figure 7 and Figure 8 show the M number and the aforementioned probability respectively for both NTZipBs and C-PackBs. In our context, the behavior of a system is decided by the worst case, because due to wear leveling technique the worst-case takes place in a number of different memory places as time passes. For this reason the average M and the average overflow probability with $M+1$ failures lack any meaning in this context and they are not shown.

First, note that the theoretical limit of failures a paired block is able to survive is 26 for NTZipBs and 30 for C-PackBs:

- In the case of NTZipBs the maximum compresses size is 576. Therefore, there are at least $1024 - 576 = 448$ bits for correcting errors. In order to correct an error, we need, at most, 8 bits for the backspace (1 bit to signal we are indexing in the table, plus 7 bits to index the table if the error is in the 64^{th} symbol). After the backspace we need to output the symbol again, which if it was not in the table previously takes 9 bits, for a total of 17 extra bits. Dividing, NTZipBs is able to correct, at least, 448/17=26 failures.
- For C-PackBs the maximum compressed size is 544. Therefore, there are at least $1024 - 544 = 480$ bits for correcting errors. In order to correct an error, we need, at most, 16 bits: 3 bits for the BS code, plus the 5 bit mask, and then 8 bits with the correction mask. In the case of errors happening in the same byte/symbol, this increases the total bits required but decreases the bits required per error. Likewise, if a failure transforms a symbol into a BS, we just discard that byte, clearing the mask, and repeat the symbol again. Dividing, C-PackBs is able to correct, at least, 480/16=30 failures.

According to Figure 7, in the case of NTZipBs a paired block is able to survive, at least, 27 failures. The theoretical limit is a strange scenario and does not show up in our simulations. Moreover, for many of the applications more than 30 failures are survived. In addition, note that this is the number of exposed failures. If a cell is stuck-at a value, and the block requires the cell to be that value, the failure is not exposed, and therefore no actions are required. This is important, because previous schemes as ECP do not take into account if the error is exposed or not, and just avoid the failing bit all the time. Moreover, if the probability of a cell holding a 0 is the same of that of a cell ending up stuck-up at 0, then, the expected number of exposed failures per block write is half the number of failures. In other words, on average, half of the stuck-at cells are written the same value they hold, and only the other half require correcting actions. This makes NTZipBs able to at least $26 * 2 = 52$ failures per paired block, which dramatically improves the errors achieved by 2 ECP blocks in their own.

However, in the case of C-PackBs the theoretical limit is actually reached for some applications, although a number of them show a more favorable behavior, managing to get more failures survived.

In the same line that happened with average compression size between NTZip and C-Pack, NTZipBs and C-PackBs do not compare in a per-application basis. There are examples of both beating the other. On the big figure, C-PackBs is able to correct, at least 30 failures, which is 4 more than the minimum for NTZipBs.

Figure 8 shows the overflow probability for NTZipBs and C-PackBs. It also includes the overflow probability for NTZip and C-Pack –calculated in an analogous fashion– in order to illustrate that for the vast majority of applications they are much higher than those of NTZipBs and C-PackBs. According to Figure 8, after the $(M + 1)^{th}$ error, the overflow probability is quite low for both schemes, but for *calculix* and *soplex*, allowing the block pair to be usable for a number of times before it actually overflows and the whole page has to be discarded. It is worth noticing that *soplex* is the application with the highest M for both algorithms, and for that, we do not think having a high overflow probability after surviving 45 or 48 failures is really a shortcoming.

To finish the high level analysis of NTZipBs and C-PackBs, we analyze for the blocks that contain failures how the amount of bits that flip varies with the number of failures. We have explored this variation on the average number of bits that flip when the number of errors grows. Owing to lack of space, we do not show the graphs, but in both schemes and for all the applications evaluated the average amount of bits flipped per block increases quasi-linearly with the number of errors. Some curves show a small bending, but it can be approximated by a horizontal line with a admissible error. In both cases all curves show a slope that ranges from around 0.3 to around 1 in the case of NTZipBs and between 0.25 and 1 in the case of C-PackBs, what means that the wear rate of the surviving cells is not vastly increased as cells start getting stuck-at. In other words, this hints that having more errors will not, necessarily, increase the rate at which cells wear. However, if we compare C-PackBs and NTZipBs in this field, C-PackBs clearly outmatches NTZipBs. Although the slopes and the general shape of the curves is the same,

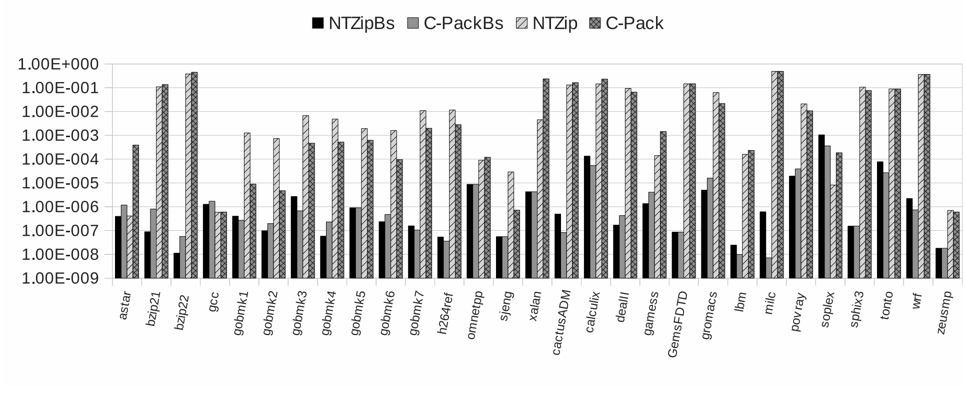


Fig. 8. Overflow probability for both NTZipBs and C-PackBs (and also NTZip and C-Pack).

those belonging to C-PackBs are at lower positions, meaning that the amount of bit-flips in a block compressed with C-PackBs is smaller than the amount of bit-flips in a block compressed with NTZip, meaning that NTZip wears the memory faster than C-PackBs does. This is mainly due to the design of the algorithms. NTZip is string-oriented, while C-Pack is word-oriented. If a byte inside a chain changes, the output is likely to vary significantly. It is likely that what was previously a chain in the dictionary now has to be compressed into two different tokens, or the other way round. On the other hand, in C-PackBs words match one of the patterns shown in Table 3. Any change in the word that does not alter which pattern the word matches will not change the output symbol encoding, maintaining the width, and the symbol header, maintaining the output largely unmodified.

5.3. *Comparison to previous proposals*

Next we provide data illustrating how well a memory system implementing our technique performs. We compare our technique to some proposals in the literature: SECDED [35], DRM [29] and ECP [26].

For a number of years, DRAM memories with Error Correcting Codes (ECC) are available. It is intuitive to use ECC for PCM before exploring other alternatives, this is why we evaluate the performance of a system implementing a SECDED (Single Error Correction, Double Error Detection) scheme, that allows to correct one error per 64-bit chunk inside a block, but when a chunk manifests a second error, its whole page is discarded from the system.

Dynamically Replicated Memory (DRM) [29] is briefly detailed in Section 6 and Error Correction Pointers (ECP) [26] is an alternative to ECC for error correction in phase change memories that uses pointers to point errors and replacement cells to survive them (see Section 2.4).

Table 6. High level comparison of schemes. The first three columns show the characteristics of each algorithm while the last two show the results of our evaluation of a system with 1000 pages using that technique. †: This amount of errors can be corrected only when the corresponding unit is paired with another such unit. When the unit is on its own this number is smaller.

Scheme	Overhead	Failure Unit	Failures/unit survived	total failures	failures per page
SECDED	10.9%	64 bits	1	29072	26.21
DRM	12.5%	4KB page	$160^{\dagger}$	161216	149.78
ECP_6	12.5%	512 bits	6	149160	149.16
CEPRAM	12.5%	512 bits	$\geq 30^{\dagger}$	1446613	1447.11

5.3.1. *Overview*

Table 5.3.1 shows a comparison of all the schemes in big numbers: the overhead, the failure unit, the correcting capability per unit, the number of failures successfully corrected on the moment of system failure, and the average number of errors in pages at system failure (failed pages are expected to have one more errors than the correction capability). We can see that CEPRAM dramatically increases both numbers, hinting for a longer lifetime. By design, CEPRAM cannot do worse than ECP, but these numbers show the big increase achieved in error-correcting capability.

First thing, point out that the numbers in the last two column correspond with the quantities when all pages are broken, therefore, they do not reflect how soon/late errors manifest. There are two interesting things to observe:

(1) Theoretical/real failure survival: For SECDED, each 4KB page is made of 512 64-bit blocks, meaning that in the presence of a good leveling of the failures, up to 512 failures can be corrected, but on average only 26.21 are. This is less than a 5%, and that is a poor ratio. DRM does somehow better, getting quite close to its capacity. Note that the 160 survived errors refers to a pair of pages. ECP_6 is, again, not very good in this respect: there are 64 blocks in each page, accounting for a total 384 ECPs, but on average, only 149.16 are used, which is below 50%. To end with, CEPRAM corrects, on average, 22.611 failures per block (1447.11/64), which is close to the “at least” 30 per block pair, but not as close DRM is to its limit.
(2) The reason behind CEPRAM being closer to the theoretical limit is, mainly, because cell lifetime follows (and is modeled as) a normal distribution, therefore, lifetimes are clustered around the average. If a scheme is able to survive those failures outlaying in the curve, then the errors are *evenly spread* among blocks. One example of the opposite behavior can be found on SECDED. Given that the failure survival ability is so small, the probability of having a byte with two cells having shorter lifetime in a block is quite high, leading to early failure of pages.

This analysis shows that although the wear is spread evenly through all the memory cells, there are many failure-surviving-resources not used at the point of failure. Developing techniques to make a more efficient usage of this resources is out of the scope of this work.

5.3.2. *Dynamic analysis*

Here we show and discuss the dynamic behavior of CEPRAM, comparing it to the aforementioned techniques. Figure 9 shows curves for all the techniques. This curve containing the point (x, y) means that after x billions writes to every page (wear leveling is assumed), the y% of the memory pages are still usable. Same graph as shown in [26], although the scale changes because they assume that in each write a bit is flipped with probability $p = 0.5$. Again, given that CEPRAM is built on top of ECP, it is bound to improve it.

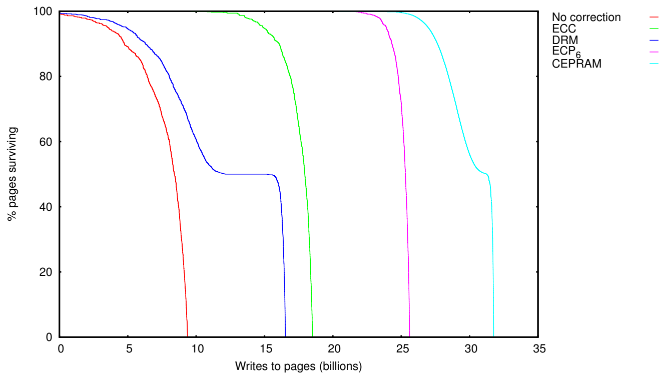
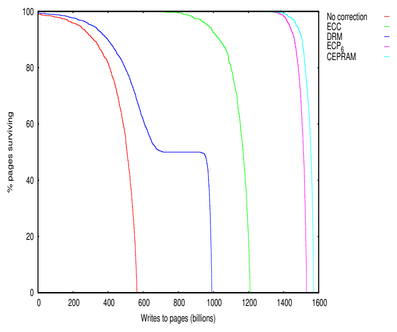
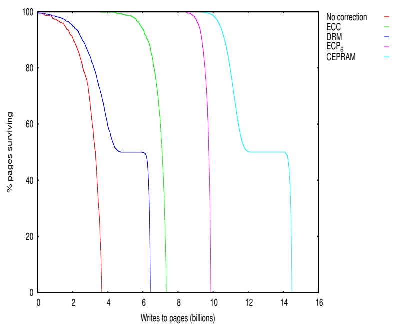


Fig. 9. % of pages functional as time (measured in blocks written back to physical pages) passes.

This graph is much more illustrative of the effectiveness of the schemes when it comes to keeping as much memory alive as possible. According to Table 5.3.1, the number of failures per page survived by ECP_6 and DRM is quite close. Nevertheless, the lifetime of a system implementing DRM is much shorter. This is due to dynamic behavior. More precisely, pages are discarded upon first failure, rapidly decreasing the memory size, making other pages *absorb* writes to those discarded pages, increasing the pace at which cells wear out, leading for an early system failure. The other techniques are somehow characterized by the appearance of the first failure, moment in which, due to the aforementioned clustering of failures when getting closer to the average cell lifetime, and to the increased wear ratio due to page failures, the amount of memory available decreases exponentially, quickly leading to a system failure. CEPRAM increases the lifetime beyond ECP due to the fine-grain pairing (as opposed to the coarse grain pairing of DRM). This pairing requires a technique to survive the existing failures. DRM uses parity to mark *dead* bytes in the primary page, CEPRAM uses compression with backspace symbol to



(a)



(b)

Fig. 10. Percentage of pages functional as the time passes for the flip/overflow probabilities of *gobmk*$_5$ (a) and *lbm* (b). Time is measured in blocks written back (to physical pages).

encode the information plus the failure information. There is a small phase in which the block is compressed with C-Pack, without backspace, encoding the bits to be discarded in a mask, prior to the pairing. Our simulations show that the worst case is frequent enough to make the length of this phase negligible, showing that the strength of CEPRAM is not as much in the compression as a means of reducing size, but as a means of encoding both the data and information about errors in the block.

The results above correspond to *dealII* which is an *average* application in terms of bit-flip probability, which is a measure of how wear-some the application is to the memory. If we take a look at *gobmk*$_5$ and *lbm*, which are the application with the lowest and highest bit-flip rate, depicted in Figures 10(a) and 10(b) respectively, we can observe that CEPRAM is especially powerful when the amount of flips is high. This is counterintuitive, because less flips mean lower entropy which usually translates in better compressibility. This happens because when the wear rate is high, cells suffer a lot of wear before an overflow happens, meaning that a lot of errors can be survived. Conversely, if the wear ratio is low, an overflow will take place before the toll is to high on the cells, not leveraging the extra error correction capacity provided by CEPRAM.

5.3.3. *Ideal case study*

To finish the evaluation, we present some discussion of the *ideal scheme* that can correct up to n errors with a 12.5% overhead (*Id* n). Figure 11 shows the available percentage of pages after a given number of writes.

Surviving the first failure almost doubles the lifetime of the system. To double that time again requires surviving 8 failures. ECP_6 does slightly better than *Id 6*. This is not weird, taking into account that failures can be hidden. For example, if

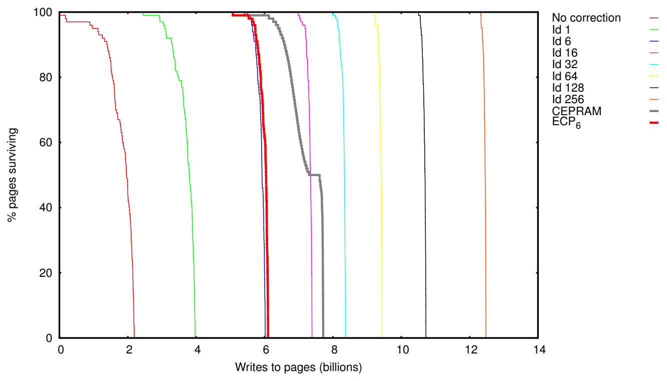


Fig. 11. % of pages functional as time (measured in blocks written back to physical pages) passes.

two cells fail, and due to wear leveling techniques they lay in the same pointer, the two errors are hiding each other, making one less pointers necessary, saving some wear. This delays the death of some pages, reducing the write-pressure over the rest of the pages, slightly spanning the lifetime. If the system failure is triggered by the available memory size falling below 50% as done in [29], CEPRAM is halfway between *16* and *32*. On the other hand, if we are more restrictive and use the ratio $1/\sqrt{2}$ *i.e.* 70.7% CEPRAM is placed between ECP_6 and *Id 16*, around $\frac{4}{5}$ of the way closer to the latter.

Figure 12 shows the accumulated number of *stuck-at* failures as a function of the lifetime. When the first failures start to happen, correcting one failure significantly improves lifetime, but as soon as we correct the 29^{th} failure, the lifetime gain per corrected failure drops down below 0.5%failure. We think that this low return of investment makes correcting more than 29 failures unprofitable. According to this reasoning CEPRAM does a great job pushing the lifetime curve towards the Id 32 curve shown in Figure 11.

6. Related work

The problem of PCM cell lifetime has been addressed mainly in three different ways. The first way is to reduce the wear experienced by cells by reducing the number of PCM writes. The first idea was in [36], and consists in detecting and avoiding silent writes. Differential writes (DCW) were introduced in [9] showing a big improvement in lifetime. Buffering techniques have also been explored in [15,16,17] presenting a level of DRAM buffering to alleviate the wear on PCM. In [17], the authors propose some techniques to overcome the process variation in terms of power and lifetime. They also use compression, but just to reduce the width of writes to some blocks that are deemed compressible by an OS-guided procedure. Flip-N-Write [11] reduces wear by flipping those words that incur in a lot of bit-flips in combination of DCW to reduce the number of effective bit-flips, thus extending the lifetime. The LLC

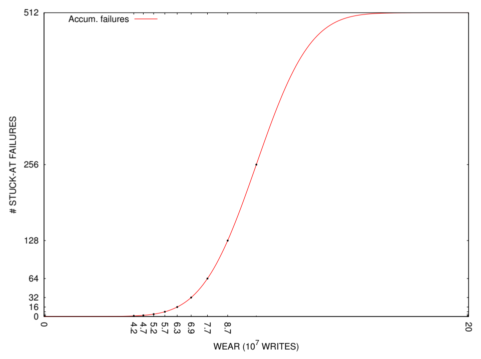


Fig. 12. Accumulated count of errors in a 512-bit block as a function of the number of writes to the block, assuming a write modifies all the block. If wear is leveled, the fact that not all the block is written merely changes the scale of the x axis.

has the ability to help in this purpose as in [37]. A modification on the replacement policy of the LLC to make it PCM-aware can help in reducing the amount of writes done to main memory without hurting performance.

Another way to tackle the limited lifetime is to evenly spread the wear among all cells [9,31,38] at different granularity, and from different perspectives: the two former are hardware techniques, and the latter is a software technique at OS level. These techniques prevent the appearance of hot spots that can lead to an early failure of the memory system.

The third trend is to detect and survive errors [29,26,39,40,41,42,43] as this piece of work does. DRM [29] does coarse grain pairing using parity to detect failing bytes. As long as two pairs have not overlapping bytes failing, they can operate as a single page tolerating some failures. ECP [26] uses $< pointer, replacement\_cell >$ pairs to point and substitute failing cells, allowing for one failure survival per pointer. FGCR [42] and PAYG [43] are techniques built on top of ECP. The former does current regulation and voltage up-scaling techniques to reduce the wear of cells, and the latter an abstraction of wear leveling applied to ECP pointers: At system failure time, a big share of the blocks have used less than 2 pointers, therefore, it would be more efficient to have a pool of pointers, and allocate them on demand as faults appear. Free-p [40] applies fine-grain pairing as CEPRAM. The main difference is that FREE-p is simple to implement and requires few extra hardware, but when a block fails it just points a “healthy” one instead of combining them somehow. On the other hand, CEPRAM is more complex because it adds a technique to survive failures that arise when two blocks are combined. SAFER [39] and RDIS [41] try to break a block down in pieces with different properties. SAFER makes n partitions, equal in size, such that there are at most one error per partition. Each partition is provided with a bit to express whether the that partition data should be flipped or not after reading. When a partition is written to, if no faults are detected, the corresponding bit is clear. If a fault is detected, then the bit is set, the word is flipped and written to the cells, so the fault is masked. RDIS does something similar. It tries to dynamically identify a set comprised only by exposed failures and healthy cells such that no exposed failures lie out of the set. Such a set is called *invertible set*. The intended contents of the invertible set are inverted and written to the hardware, avoiding the exposure of any fault, allowing for stuck-at fault survival.

As far as cache compression is concerned, other than compression algorithms as C-Pack [23], some work has been done on architecture and interfaces for systems with compressed memory systems [24].

Our proposal is the first one to use compression for endurance, extending the lifetime of a PRAM based system. It builds on top of ECP as some other proposals, and takes the lifetime well beyond the limit of ECP with a novel way of dealing with exposed errors through the use of a custom backspace encoding to ignore (NTZipBs) or repair (C-PackBs) exposed failures.

7. Conclusions

Resistive memories are closer to industrial adoption every day. That makes it important to develop techniques to overcome the limitations they present, so they can substitute DRAM, improving the memory in aspects such as scalability, low leakage.

In this paper we have introduced CEPRAM, an attempt to make a PRAM system more durable through memory block compression. Our technique is built on top of ECP, and using a high-performance, cache-oriented compression algorithm, modified to better suit our purpose. It manages to further extend the lifetime of the memory system. In particular, it guarantees that at least half of the physical pages are in usable condition for 25% longer than *ECP*, which is slightly more than 5% more than an scheme that can correct 16 failures per block.

In addition , we do an analysis of why PCM is not a very good context for compression, and it is probably not a good idea to invest efforts in it, unless compression is conceived differently: focusing more on improving the worst case scenario.

Finally, we show an study on the memory life time improvement as a function of the error correcting ability, offering some insights to help choose a target failure recovery ability depending on the lifetime expansion we want to get. This study shows that there is room for improvement beyond CEPRAM, although CEPRAM pushes the lifetime to a point in which the cost of extending the lifetime any further is quite high in terms of the amount of per-block error correcting ability.

References

1. K. Kim, Technology for sub-50nm dram and nand flash manufacturing, in *International Electron Devices Meeting 2005*, (Washington, 2005), pp. 323–326.
2. Y. Chen, X. Wang, H. Li, H. Liu and D. Dimitrov, Design margin exploration of spin-torque transfer ram (spram), in *ISQED*, 2008, pp. 684 –690.

3. A. Ruan, B. Hu and Y. Zhai, A parasitic effect - free test scheme for ferroelectric random access memory (fram), in *Int. Conf. on Testing and Diagnosis*, 2009, pp. 1–4.
4. A. P. Ferreira, M. Zhou, S. Bock, B. R. Childers, R. G. Melhem and D. Mossé, Increasing PCM main memory lifetime, in *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12*, 2010, pp. 914–919.
5. X. Zhang, Q. Hu, D. Wang, C. Li and H. Wang, A read-write aware replacement policy for phase change memory, in *Advanced Parallel Processing Technologies*, (Springer, 2011) pp. 31–45.
6. S. Yoo, E. Lee and H. Bahn, LDF (less dirty first): dirtiness-aware cache replacement policy for PCM main memory, *Electronics Letters* **49**(25) (2013) p. 1607.
7. R. Rodríguez-Rodríguez, F. Castro, D. Chaver, L. Piñuel and F. Tirado, Reducing writes in phase-change memory environments by using efficient cache replacement policies, in *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, 2013, pp. 93–96.
8. W. Zhang and T. Li, Characterizing and mitigating the impact of process variations on phase change based memory systems, *Proceedings of the Micro-42* (2009) 2–13.
9. P. Zhou, B. Zhao, J. Yang and Y. Zhang, A durable and energy efficient main memory using phase change memory technology, in *Proc. of ISCA*, 2009, pp. 14–23.
10. B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu and D. Burger, Phase-Change Technology and the Future of Main Memory, *IEEE Micro* **30**(1) (2010) 131–141.
11. S. Cho and H. Lee, Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance, in *Proc. of MICRO.*, 2009, pp. 347–357.
12. L. E. Ramos, E. Gorbatov and R. Bianchini, Page placement in hybrid memory systems, in *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04*, ACM2011, pp. 85–95.
13. I.-S. Choi, S.-I. Jang, C.-H. Oh, C. C. Weems and S.-D. Kim, A dynamic adaptive converter and management for PRAM-based main memory, *Microprocessors and Microsystems* **37**(6) (2013) 554–561.
14. M. K. Qureshi, S. Gurumurthi and B. Rajendran, Phase change memory: From devices to systems, *Synthesis Lectures on Computer Architecture* **6**(4) (2011) 1–134.
15. M. K. Qureshi *et al.*, Scalable high performance main memory system using phase-change memory technology, in *Proc. of ISCA*, 2009, pp. 24–33.
16. B. C. Lee, E. Ipek, O. Mutlu and D. Burger, Architecting phase change memory as a scalable dram alternative, *SIGARCH Comput. Archit. News* **37**(June 2009) 2–13.
17. W. Zhang and T. Li, Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures, in *Proc. of Int. Conference on Parallel Architectures and Compilation Techniques*, *PACT*2009, pp. 101–112.
18. S. Lai, Current status of the phase change memory and its future, in *International Electron Devices Meeting 2003*, (Santa Clara, CA, 2003), pp. 255–258.
19. R. F. Freitas and W. W. Wilcke, Storage-class memory: The next storage system technology, *IBM Journal of Research and Development* **52**(4.5) (2008) 439–447.
20. T. Nirschl *et al.*, Write strategies for 2 and 4-bit multi-level phase-change memory, in *IEEE Int. Electron Devices Meeting, 2007. IEDM 2007*, 2007, pp. 461 –464.
21. S. Mukherjee, J. Emer and S. Reinhardt, The soft error problem: an architectural perspective, in *HPCA-11.*, 2005, pp. 243 – 247.
22. J. F. Ziegler, Terrestrial cosmic rays, *IBM Journal of Research and Develop.* **40**(1) (1996) 19–39.
23. X. Chen, L. Yang, R. Dick, L. Shang and H. Lekatsas, C-pack: A high-performance microprocessor cache compression algorithm, *IEEE Transactions on VLSI Systems,*

18(8) (2010) 1196 –1208.
24. C. Benveniste, P. Franaszek and J. Robinson, Cache-memory interfaces in compressed memory systems, *IEEE Transactions on Computers,* **50**(11) (2001) 1106 –1116.
25. P. Pujara and A. Aggarwal, Restrictive compression techniques to increase level 1 cache capacity, in *Int. Conference on Computer Design (ICCD).*, 2005, pp. 327 – 333.
26. S. Schechter, G. H. Loh, K. Straus and D. Burger, Use ecp, not ecc, for hard failures in resistive memories, in *Int. Symposium on Computer Arch. (ISCA)*, 2010, pp. 141–152.
27. T. A. Welch, A technique for high-performance data compression, *Computer* **17**(June 1984) 8–19.
28. R. Hamming, Error Detecting and Error Correcting Codes, *Bell System Technical Journal* **26**(2) (1950) 147–160.
29. E. Ipek *et al.*, Dynamically replicated memory: building reliable systems from nanoscale resistive memories, in *ASPLOS*, Mar. 2010.
30. C. Wilkerson *et al.*, Trading off cache capacity for reliability to enable low voltage operation, in *Int. Symposium on Computer Architecture*, *ISCA*.2008, pp. 203–214.
31. M. K. Qureshi *et al.*, Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling, in *Proc. of MICRO*, 2009, pp. 14–23.
32. T. S. P. E. Corporation, The spec cpu 2006 benchmark suite (2013), http://www.specbench.org.
33. J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prulovic., L. Ceze, S. Sarangi, P. Sack, K. Strauss and P. Montesinos, Sesc simulator (2013), http://sesc.sourceforge.net.
34. C.-K. Luk *et al.*, Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, in *Proc. of PLDI*, 2005, pp. 190 – 200.
35. M. Y. Hsiao, A class of optimal minimum odd-weight-column sec-ded codes, *IBM J. Res. Dev.* **14**(July 1970) 395–401.
36. B.-D. Yang *et al.*, A low power phase-change random access memory using a data-comparison write scheme, in *Proc. of ISCAS*, may. 2007, pp. 3014 –3017.
37. M. Zhou *et al.*, Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems, *ACM Trans. Archit. Code Optim.* **8**(4) (2012) 53:1–53:21.
38. N. H. Seong, D. H. Woo and H.-H. S. Lee, Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping, in *Proceedings of ISCA*, 2010, pp. 383–394.
39. N. H. Seong, D. H. Woo, V. Srinivasan, J. Rivers and H.-H. Lee, Safer: Stuck-at-fault error recovery for memories, in *Proc. of Int. Symposium on Microarchitecture (MICRO),*, dec. 2010, pp. 115 –124.
40. D. H. Yoon *et al.*, Free-p: A practical end-to-end nonvolatile memory protection mechanism, *Micro, IEEE* **32**(3) (2012) 79 –87.
41. R. Melhem, R. Maddah and S. Cho, Rdis: A recursively defined invertible set scheme to tolerate multiple stuck-at faults in resistive memory, in *Proc. of IEEE/IFIP International Conference on Dependable System and Networks (DSN)*, 2012.
42. L. Jiang, Y. Zhang and J. Yang, Enhancing phase change memory lifetime through fine-grained current regulation and voltage upscaling, in *Proc. of Int. Symposium on Low-power electronics and design*, *ISLPED*2011, pp. 127–132.
43. M. K. Qureshi, Pay-as-you-go: low-overhead hard-error correction for phase change memories, in *Proc. of MICRO*, 2011, pp. 318–328.