Memory Carousel: LLVM-Based Bitwise Wear-Leveling for Non-Volatile Main Memory

Nils Hölscher, Christian Hakert, Hassan Nassar, Kuan-Hsun Chen *Member, IEEE*, Lars Bauer *Member, IEEE*,
Jian-Jia Chen *Senior Member, IEEE*, and Jörg Henkel *Senior Member, IEEE*

***Abstract*— Emerging non-volatile memory yields, alongside many advantages, technical shortcomings, such as reduced cell lifetime. Although many wear-leveling approaches exist to extend the lifetime of such memories, usually a trade-off for the granularity of wear-leveling has to be made. Due to iterative write schemes (repeatedly sense and write), wear-out of memory in certain systems is directly dependent on the written bit value and thus can be highly imbalanced, requiring dedicated bit-wise wear-leveling. Such a bit-wise wear-leveling so far has only be proposed together with a special hardware support. However, if no dedicated hardware solutions are available, especially for commercial off-the-shelf systems with non-volatile memories, a software solution can be crucial for the system lifetime.**

In this work, we propose entirely software-based bit-wise wear-leveling, where the position of bits within CPU words in main memory is rotated on a regular basis. We leverage the LLVM intermediate representation to adjust load and store operations of the application with a custom compiler pass. Experimental evaluation shows that the lifetime by applying local rotation within the CPU word can be extended by a factor of up to $21\times$. We also show that our method can incorporate with coarser-grained wear-leveling, e.g. on block granularity and assist achievement of higher lifetime improvements.

***Index Terms*—Wear-Leveling, Non-Volatile Main Memory, Intermediate Representation, LLVM, Bit Rotation**

I. Introduction

DUE to the widely realized implementation of iterative write schemes [1]–[3] in emerging non-volatile main memory (NVM), wear-out of such memories becomes highly non-uniform even on the bit granularity. When applying iterative write schemes, memory cells are sensed before every write operation and only adequate write pulses are applied in an iterative manner until the target cell value is reached. As a result, writing a memory cell with the value it contained before, causes no wear-out while changing the cell value causes memory wear-out. On a single-level cell memory, a single bit is stored per memory cell, thus cells wear-out from bit flips only.

In order to accommodate for limited memory lifetime, a broad landscape of wear-leveling methods for NVM is explored in the literature [4]–[10]. The majority of these methods considers memory wear-out to happen uniformly within blocks of a certain granularity (e.g. words, cache-lines, memory pages) and therefore wear-levels such entire blocks. Hence, these methods do not accommodate for iterative write scheme memories. Considering a simple example of a 64 bit counter variable, increased by one each time its written, a uniform wear-out assumption would consider all 8 bytes to be written on every update of the variable and cause wear-out. Indeed, when only incrementing the counter, the least significant bit is updated every time and causes wear-out. Each higher significant bit is only updated half as often as the next lower significant bit, thus a logarithmic distribution of wear-out is caused within the 8 bytes. While it may seem unrealistic that all memory content of a program behaves similar like such variables, an initial case study reveals a way higher wear-out within the lower significant bits compared to the higher significant bits for a broad range of typical benchmark applications [11].

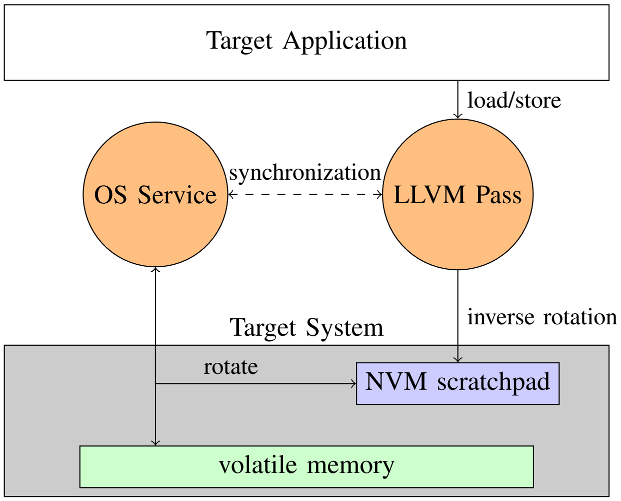


Fig. 1. Overview of Memory Carousel, where the solid arrows show the stages passes by the applications memory accesses. The dashed arrow visualizes the need of synchronization between the two provided services in this work.

Motivated by this observation, we investigate the *problem* of wear-leveling on a bit granularity in order to accommodate for iterative write scheme memories in this paper. While several solutions exist to perform such wear-leveling with special hardware support [4], [12], in this work, we propose our *problem solution* – **Memory Carousel**, which is an entirely software-based wear-leveling method for iterative write

N. Hölscher, C. Hakert, and J.-J. Chen are with the Design Automation for Embedded Systems Group, TU Dortmund University, Germany. Email: {nils.hoelscher, christian.hakert}@tu-dortmund.de, jian-jia.chen@cs.uni-dortmund.de

K.-H. Chen is with the Chair of Computer Architecture and Embedded Systems (CAES), University of Twente, the Netherlands. Email: k.h.chen@utwente.nl

H. Nassar, L. Bauer, and J. Henkel are with the Chair for Embedded Systems (CES), Karlsruhe Institute of Technology, Germany. Email: {hassan.nassar, lars.bauer, henkel}@kit.edu

scheme memories. The main idea is to rotate the physical position of logic bits within memory words continuously in the operating systems and simultaneously preserve the correctness of program execution on the rotated memory space with the support of bit shift operations in the compiler pass.

Our novel contributions: Figure 1 illustrates the overview of Memory Carousel. Basically, we provide two services, i.e., one in OS and another one in the LLVM compiler, and the key component of our approach is to maintain the correctness of program execution on the rotated memory space. We compile the target application to LLVM intermediate representation, patch all load and store operations with special inverse rotation code, and ultimately compile the application to machine code. In a nutshell, our contributions can be listed as follows:

- An operating system service to continuously rotate a certain memory region, denoted as memory interval, in order to move the physical position of highly worn-out bits to the entire memory space.
- An LLVM pass that extends load and store operations by inverse rotation code and therefore maintains the correctness of data load and store and therefore of the application execution.
- A lightweight synchronization scheme between the operating system component and the LLVM code to avoid race conditions during rotation of the memory.
- A Valgrind-based offline profiling tool, approximately simulating our wear-leveling method on a given target application in order to allow a trade-off decision between lifetime improvement and caused overheads.

Extensive evaluation with a full system simulation allows precise analysis of the improvement of memory lifetime with respect to the iterative write scheme and the caused overheads. We show that we can improve the memory lifetime by up to a factor of 21×, and that we are able to identify applications, where our method causes higher overheads than improvements upfront, by using our Valgrind-based profiling tool. The corresponding source code is ready to be released open source.

II. System Model

As a target system for our method, we assume small systems with application processors in this work. The considered system can be equipped with classic volatile memory (e.g. DRAM) and with additional non-volatile memory. In this paper, we focus on storing application specific data structures within NVM. Storing the allover infrastructure (i.e. the operating system, drivers, stack, etc.) is beyond the scope of this work. The system software can decide to load certain memory contents to the NVM in order to provide persistence. In this work, we assume that the target application is loaded with the full memory footprint to such an NVM scratchpad and the operating system and system software resides separately in volatile memory. We further consider the NVM to be a scratchpad memory, which is usually small (e.g. few hundred kilobytes) and fast, and we assume it is not covered by further caches, thus all memory requests directly go to the NVM.

As motivated by [1]–[3], in this work, we focus on iterative write scheme memories, i.e., cells are only written when the cell value is changed. Hence, the memory hardware reads out the cell value prior to an update and only applies an adequate update operation to the cell in an iterative manner. We assume single-level cell, i.e., one bit corresponds to exactly one memory cell. That is, the wear-out of each memory cell is linearly related to the amount of bit flips in the memory cell. If multi-level cells are used, analysis of the wear-out is still possible but requires more detailed modelling, since not all changes of a cell value may cause the same wear-out, which is considered out of scope in this work.

Our implementation provides a custom synchronization mechanism, which relies on memory access permissions and memory permission violation traps, which we assume to be provided by an MMU. However, the implementation can be straightforward adopted to another synchronization scheme, which does not depend on the presence of an MMU. In this paper, the proposed methods are implemented as a real system service in a simulation system [13]. As the simulated system of this setup, gem5 [13] is configure to to run the VExpress_GEM5_V2 machine, with a DerivO3 ARMv8 64 bit CPU. This configuration corresponds to an ARMv8 application processor (e.g. in desktop PCs or powerful embedded systems), including, among others, multiple cores, pipelining and out-of-order execution.

III. Bit-Wise Memory Wear-Out

Technical realizations of emerging non-volatile memory bring up various schemes for managing read and write accesses to the memory. One dedicated scheme is the iterative write scheme [1]–[3]. If a cell already contains the target value, the cell is not updated at all. For the other cells, write pulses are applied in iterative steps until they reach the target value. Applying this method can help to reduce latencies, energy consumption, and even the total memory wear-out, since cells are not unnecessarily stressed. The wear-out, however, becomes less uniform, since some bits of the memory may be flipped more often than others. In consequence, if the memory lifetime should be extended, *the uneven wear-out of single bits* needs to be well leveled and spread across all other bits. In this section, we illustrate the problem by investigating a concrete example and present means to quantify the problem.

A. Memory Age Analysis

First, we investigate the uneven amount of bit flips within a programs memory interval. In this work, we adopt full system simulations [14], where we can assess the memory content before and after a write operation in order to determine the bit flips per memory cell. Since the iterative write scheme is assumed, the wear-out cannot be determined by investigating the amount of write accesses to a certain memory location directly as the built-in approach. The memory contents rather have to be investigated and it has to be determined if the write access causes a bitflip in a certain cell or not. We extend the simulation environment so that collected data can be further processed and indicators about the possible lifetime extension of the memory can be computed.

Assuming that the wear-out could be ideally spread within words, we compute the **Achieved Endurance** $AE_{p(i)}$ of a program p and its implementation i in memory interval I. This is achieved by measuring all bit flips from a start address s to an end address e, with $I = [s, e]$. The number of bit flips produced by $p(i)$ over I shall be called flip_count.

$$AE^I_{p(i)} = \frac{mean(\text{flip\_count})}{max(\text{flip\_count})} \tag{1}$$

This effectively provides a metric indicating the quality of wear-leveling within I, during a programs $p(i)$ execution. A memory interval could be ideally wear-leveled, if bit flips are redirected in a way, that all bits face exactly the same amount of flips, i.e. the mean amount of bit flips equals the maximum amount of flips. Without adding additional fresh memory, lifetime could not be further improved. Assuming that the memory becomes unusable once the first bit dies, the relation between the mean and max flip count is the fraction of the ideal memory lifetime achieved. An AE of 1 means that all bit flips are evenly distributed and no further improvements can be made. An AE of 0.5, for instance, means that the lifetime can be doubled with ideal wear-leveling.

The achieved endurance of a program's execution can be further compared to another implementation of the program, with applied wear-leveling, with $p(wl)$. The run without wear-leveling is the base-line run $p(b)$. These two runs can now be compared in regards to the introduced **Overhead** OV, **Endurance Improvement** EI, and the **Lifetime Improvement** LI.

The OV describes how many bit flips are introduced in addition to the base run. When $OV = 1.45$, this means that the wear-leveled run introduces 45% more bit flips compared to the base run.

$$OV^I = \frac{\sum_i^I \text{flip\_count}^i_{p(wl)}}{\sum_i^I \text{flip\_count}^i_{p(b)}} \tag{2}$$

Equation (2) computes the caused overhead (OV), by summing up the total amount of bit flips across all intervals for a baseline run and a run with wear-leveling and building the fraction between both. Thus, the additional bit flips, caused by the wear-leveling are reported in this overhead calculation.

A wear-leveled run should increase its AE in comparison to its base run. This improvement is represented by the EI metric. The larger EI, the better is the analysed wear-levelling approach.

$$EI^I = \frac{AE^I_{\text{p(wl)}}}{AE^I_{\text{p(b)}}} \tag{3}$$

The improvement in endurance is a metric to compare different wear-levelling approaches. However, it does not take the introduced overhead into account.

$$LI^I = \frac{EI^I}{OV^I} \tag{4}$$

Therefore the lifetime improvement LI is introduced, representing the actual lifetime increase of a memory module.

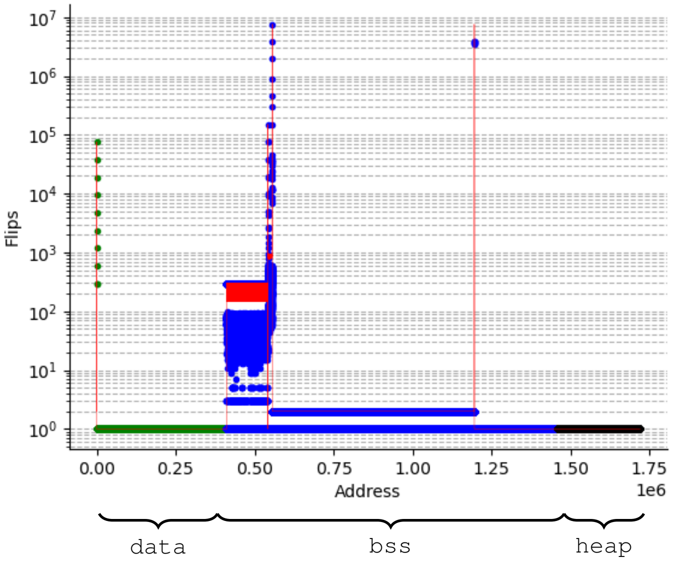


Fig. 2. Comparison of logical write accesses (red) and real bit flips (Data: green, BSS: blue, Heap: black). The x-axis shows normalised Bit addresses and is scaled by 10^6. The y-axis shows the number of bit flips.

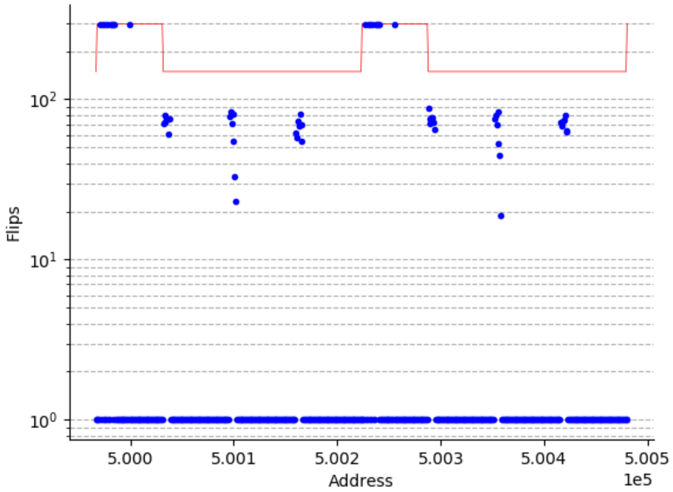


Fig. 3. A zoom-in portion of Fig. 2 at $5e5$ and onward, to show how bit flips are distributed over a memory size of 512 bits.

B. Initial Case Study

To provide intuition and to motivate the need for the aforementioned possible improvement that can be gained when iterative write schemes and bit-wise wear-leveling are applied, we consider the memory portion of the dijkstra benchmark application [13]. Figure 2 depicts the analyzed amount of write accesses to memory cells after a full system simulation of the benchmark. The red line indicates the total amount of logical write accesses (no iterative write scheme). The points indicate the real number of bit flips per memory cell, where green means `data`, blue means `bss`, and black means `heap`. Although accesses are always the same for each memory word, however this is not the case for bit flips, as shown in the zoomed in part of Fig. 2 in Fig. 3.

Firstly it can be observed that for many memory cells, the number of real bit flips is by orders of magnitude smaller than the number of write accesses. However, it can also be observed that for some memory cells, the peak number of bit flips is very close to the number of write accesses. By calculating the shortest paths between nodes in a graph, the Dijkstra algorithm has an Achieved Endurance of $AE = 3.2e^{-6}$. As mentioned before, a small AE indicates room for improvement, since it compares the actual wear-out against an theoretical optimal wear-leveled wear-out. Hence, an $AE = 3.2e^{-6}$ implies that at least one peak exists, that is 10^6 times larger compared to the theoretical ideal wear-leveled bitflip distribution.

IV. Memory Carousel: Bit Rotation

Performing wear-leveling on bit granularity at the hardware level has been discussed in the literature [4], [12]. Realizing this at the software level, however, is rarely considered. In this work, we present a method, named **Memory Carousel** to perform bit-wise wear-leveling at the software level, not to compete with hardware solutions, but to allow for an alternative when hardware solutions are not available. Our method performs wear-leveling by rotating words, i.e., 64 bits. This spreads the high and non-uniform stress of single bits equally to all bits within the word.

Two major components are developed to achieve wear-leveling on a specific memory interval. As shown in Figure 1, these components (orange) are set in context with the target system. The first one is an operating system service, continuously rotating memory words in the targeted memory interval. This service can be triggered by a memory trap, as shown in this work, or by any other triggers, e.g. a timer or some other external interrupts. The second component is an LLVM pass, patching all memory accesses in the target application and therefore guaranteeing correct execution with rotated memory words. The pass not only restores loaded data and applies the rotation to stored data, but also applies the rotation selectively on nearly arbitrary memory intervals. Therefore, the wear-leveled region can be chosen freely.

A. Rotation Operation

Figure 4 illustrates the design principle of the rotation operation. In order to realize such a rotational wear-leveling, two steps are required: 1) regular rotation of the memory content and 2) modification of the executed program to anticipate the memory rotation. While 1) is rather straight forward, 2) draws a major challenge. The executed application has to be modified to not just load and process memory contents, but to load the memory content, undo the rotation (in the following called 'unrotation') in order to retrieve the correct value, and then to process the result.

This rotation and unrotation could be introduced at various levels: The application source code could be directly modified by, for instance, only allowing special data types that perform the unrotation. The application source code could also be rewritten by a pre-processor before compiling, which would require an extension of the programming language (e.g., C/C++). An alternative approach would be to post-process the assembly code that is generated by the compiler. Memory instructions (e.g., load and store instructions) could be replaced by a code block that performs the unrotation in place. This solution, however, would become architecture dependent and possibly sophisticated, since a wide variety of memory access instructions may exist. A hybrid solution, which we apply in our method, is to modify the intermediate language during the compilation process. Hereby, we rewrite LLVM-intermediate representation (IR) code, which requires a very limited language support, since LLVM-IR only includes one type of load and store instructions. Furthermore, in LLVM-IR, we are independent of the underlying CPU architecture and assembly language.

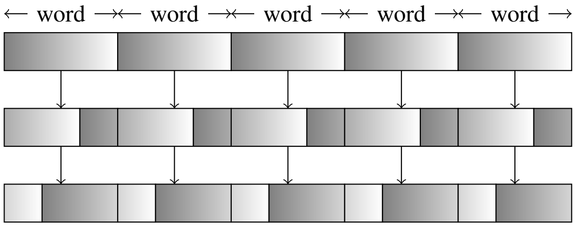


Fig. 4. Illustration of bitwise rotation of memory words. The grayscale indicates the wear-out. Rows visualize different amounts of rotation, and overlapping them will result in a more evenly spread wear-out.

B. Memory Access in LLVM-IR

The idea of an intermediate representation is to represent all target architectures a compiler can handle, while being as close to machine code as possible. LLVM-IR implements a store and a load instruction. These two instructions are the only ones writing and reading from memory. Which is highly advantageous in contrast to assembly code, where many different kinds of read and write instructions exist. The way memory is abstracted in LLVM-IR has one major drawback. It does not implement a bounded Registers-Model, therefore the number of registers are arbitrarily large. Modern compilers use register allocation to map intermediate values to machine registers. During this process values are pushed on the stack, once all registers are used and alive values still exist. Those operations are called spill and fill operations. An optimal register allocation generates the smallest possible number of spills and fills to implement the given program in its target assembly. Resulting in the stack being partially abstracted away in LLVM-IR. Thus, our proposed method to rotate memory word in the IR level can not cover the stack.

Parts of the stack are still covered by LLVM-IR like function parameters and function calling for example are covered by LLVM-IR. This raises the issue that memory accesses in LLVM-IR have to distinguish between stack accesses and other memory regions for our approach.

C. Rotated memory Load and Store

For all patched loads and stores, we assume a global variable exists, containing the current rotation amount. The code block that replaces all load or store operations in the original code are presented on a high level in Algorithms 1 and 2, where the operators $\lll$, $\ggg$ are left/right rotation operations and $\ll$, $\gg$ are left/right logic shifts with zero fills. Figure 5 illustrates

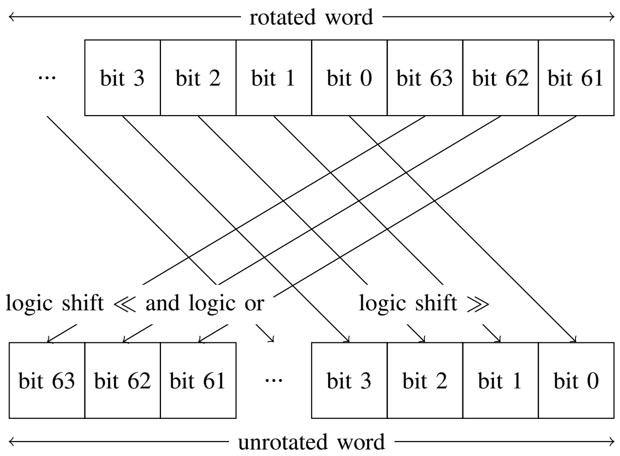


Fig. 5. Illustration of the rotation operation $\ggg$

how the rotation operations can be realized with logic shift and or operations.

Algorithm 1
Loading a N-Bit value from a 64-Bit rotated memory word

```
 1: Given: address $p*$, rotation amount at $rot*$ and $s*$, $e*$ as
    the memory interval borders
 2: $\mathit{Offset} \leftarrow p * \bmod 8$
 3: $p*_{aligned} \leftarrow p* - \mathit{Offset}$
 4: $s \leftarrow$ Load $s*$
 5: $e \leftarrow$ Load $e*$
 6: if $p* \in [s, e]$ then
 7:     $rot \leftarrow$ Load $rot*$    ↯ Critical Load
 8:     $Word_{rot} \leftarrow$ Load $p*_{aligned}$
 9:     $Word \leftarrow Word_{rot} \ggg rot$
10:     $Offset_{Bit} \leftarrow Offset * 8$
11:     $Value_{64Bit} \leftarrow Word \ggg Offset_{Bit}$
12:     $Value \leftarrow$ Truncate $Value_{64Bit}$ to $N$-Bits
13: else
14:     $Value \leftarrow$ Load $p*$
15: end if
```

Load: The Offset of the load address $p*$ is calculated first in case the pointer is not 8-byte aligned. The calculation assumes a byte-addressable memory. Next the upper and lower bounds of the rotated memory region are loaded, if the address is within this region, the word has to be rotated, otherwise the value is loaded as before. In case the value has a datatype smaller than 64 bits, it is also rotated to the front of the memory word, so that the bits not containing the value can be truncated.

Store: In contrast to the rotated load, storing a value in a memory word requires more steps. Again the offset is computed and applied to the memory region, when the address is checked. If the address is in the rotated memory section, the 64-bit word is loaded and rotated similarly as for the load. Algorithm 2 differs from the load after line 12, where the loaded word is shifted left and right by the values bit width to fill the old value with zeros. Afterwards, a bit-wise logic OR operation can be applied to the word and the zero-extended value. This results in a word with the new value at the beginning. In case the value was not stored at the beginning of the word, it has to be rotated back in place by the calculated offset, where the rotation can be applied and the word is stored.

Please note that the whole offsetting related rotation and truncating can be skipped for 64-bit data types. This is possible, because the LLVM-IR is type aware, so patches for such data types in fact consist of fewer instructions, which leads to reduced computational and program size overheads. Hence, Lines $2-3$ and $10-12$ in Algorithm 1 can be omitted, and Lines $2-3$ and $10-16$ in Algorithm 2 can also be omitted. In addition, LLVM-IR does not implement a rotation operand. Thus, all rotations consist of a left shift, a right shift and a bit-wise OR operation, as shown in Fig. 5.

Algorithm 2
Storing a N-Bit value to a 64-Bit rotated memory word

```
 1: Given: address $p*$ and its value $p$, rotation amount at $rot*$
    and $s*$, $e*$ as the memory interval borders
 2: $\mathit{Offset} \leftarrow p * \bmod 8$
 3: $p*_{aligned} \leftarrow p* - \mathit{Offset}$
 4: $s \leftarrow$ Load $s*$
 5: $e \leftarrow$ Load $e*$
 6: if $p* \in [s, e]$ then
 7:     $rot \leftarrow$ Load $rot*$    ↯ Critical Load
 8:     $Word_{rot} \leftarrow$ Load $p*_{aligned}$
 9:     $Word \leftarrow Word_{rot} \ggg rot$
10:     $Offset_{Bit} \leftarrow Offset * 8$
11:     $Word_{align} \leftarrow Word \ggg Offset_{Bit}$
12:     $Word \leftarrow Word \ll N$
13:     $Word \leftarrow Word \gg N$
14:     $p_{64} \leftarrow$ Zero extend $p$ to 64
15:     $Word_{p} \leftarrow Word \mid p_{64}$
16:     $Word_{p,align} \leftarrow Word_{p} \lll Offset_{Bit}$
17:     $Word_{p,rot} \leftarrow Word_{p,align} \lll rot$
18:     Store $Word_{p,rot}$ in $p*$
19: else
20:     Store $p$ in $p*$
21: end if
```

D. Repetitive Memory Rotation

In the previous sections, we have introduced our method to patch load and store operations in LLVM-IR to safely access rotated memory under the assumption that a global variable exists that holds the rotation amount, i.e., by how many bits a value shall be rotated. However, this method does not level the wear of memory (as the rotation amount is not changed), but 'only' ensures the correct execution of the program on rotated memory. In the following, we present an interrupt-safe solution for changing the rotation amount and rotating the designated memory during program execution.

To level wear-outs over an entire memory word (64 Bit), during execution, we aim to rotate it at least 63 times within a certain time period (e.g. several hours). Since rotating introduces overhead and we target small applications, rotating at least 63 times once during the application execution should be the ideal compromise between wear-levelling and overhead. However, this might not be the case for lager applications. To ensure this, we run the patched program once without memory rotation and count all X store accesses. This number X could also be approximated with offline analysis (e.g. static analysis or with a performance monitoring tool). In order to initiate the rotation of the memory from software, we use a write counter that triggers an overflow trap when the performance counter register overflows. The performance counter register is set to $2^{64} - (X \div 64)$, ensuring 63 rotations during program execution. This trap causes a function to iterate over the target memory locations and load 64 bit words into registers, rotate them by one bit, and store them again. However if this rotation is applied immediately, rotation could occur during a critical section. All sections between loading a memory word and loading the rotation amount can be regarded as critical. When a rotation trap is triggered between these two loads, the resulting value is off by one rotation leading to undefined program behaviour.

To guarantee the correctness of patched program execution, we have to synchronize this rotation trap with all patched load/store operations. To solve this, we employ a specific mechanism here to reduce the overhead. The current rotation offset is stored in a global variable that is read by every patched load/store operation exactly once and at the beginning in the patched code. This is done before the memory word is rotated. These critical loads are marked with ↯ in Algorithms 1 and 2. On the performance counter register overflow trap, we set the memory permissions of this variable to not allow any access. Thus, load/store operations which already read the variable still can continue and load the memory with the old rotation offset.

Once unrotation operations are completed and the next operation is about to start, it causes a trap while loading the rotation offset. Within this trap handler, we rotate the entire memory and update the rotation offset variable. After the trap handler finishes, the application repeats the load of the offset variable and sees a consistent offset variable and rotated memory. This implementation assumes that the rotated memory is only accessed by one task. However, the concept can be straight forward extended to a multicore system. In such a scenario, all cores have to cause the trap of accessing the rotation variable before the rotation and update of the variable can be triggered. In addition to a slightly increase of the time overhead due to busy waiting, this can potentially cause deadlocks, which have to be prevented.

E. Extensions with Coarser Wear-Leveling Approaches

The concept of bit-wise wear-leveling, as introduced in the previous sections, is intended to wear-level uneven bit usage within CPU words (e.g. 64 bits). During a program execution, such uneven bit wear-out can easily occur, as already shown in the case study. However, beyond the granularity of single bits, larger memory blocks itself may be also unevenly used. If for instance, multiple contiguous words in memory belong to the same logical data object, the bits within the words may be uneven used due to the written values, but the object itself may be used on another frequency than other objects.

To account for this, our bit-wise wear-leveling is designed in a fashion to work side by side with other, coarser grained wear-leveling mechanisms. Such mechanisms usually work on a memory address level, change the physical position of memory contents from time to time, and adjust the memory accesses accordingly in order to maintain correctness. As one candidate for such wear-leveling, we also study how our proposed bit-wise wear-leveling works along with small block wear-leveling approaches. Although we do not consider caches, we study an existing method, working on cache-line granularity [13]. Please note that our strategy is compatible to any arbitrary sized blocks, e.g., [12], [15]–[17].

We assume blocks of a fixed width, 64 bytes. In addition, we assume that words within each block are offsetted by one word within the block on regular intervals. Words always stay within their block and wrap around at the end and are shifted to the beginning of the block. Since full system simulations are adopted, we include a simulation of such blocks wear-leveling based on the memory trace. The simulation then is independent of whether the method would be realized in hardware or software. Please note that, we do not assess the introduced additional overheads of the technical realization. Instead, we show how well our bit-wise wear-leveling can work along with such coarser-grained methods in the next section.

V. Valgrind Profiler: Pre-Analysis

Depending on the application, bit usage within single words can be highly uniform or non-uniform. In case the usage is not uniform, Memory Carousel is able to achieve significant lifetime improvements of underlying non-volatile memory. If, however, the bit usage of the application is already nearly uniform, our bit-rotation approach cannot gain much improvements and possibly even diminishes memory lifetime due to the introduced overheads. Therefore, it is crucial to estimate in advance, whether it is beneficial to apply our method. To this end, we develop a pre-analysis method based on Valgrind, and propose an indicator called Pseudo Endurance, which is detailed in the following. Although the memory traces from our Valgrind tools only approximate the real memory trace, the relations between intensively flipped bits and less intensively flipped bits are represented and can be assessed.

A. Valgrind-based Profiling Tool

The Valgrind-based profiling tool performs the pre-analysis of the program throughout its subtool Lackey [18], which outputs the traces for the different load and store instructions performed by the program and their corresponding addresses. The developed tool focuses on the store operations that are of interest. It builds a histogram for the addresses and the number of store operations performed on them. It also provides a histogram for the bitwidths and the store operations performed.

This tool allows to derive an approximate memory trace by executing the program on almost native speed. Therefore, every application can be executed for a short time-frame (e.g. several minutes) and a distribution of uneven bit usage within words can be recorded. Based on this recording, a threshold can be defined if bit wear-leveling should be applied or not.

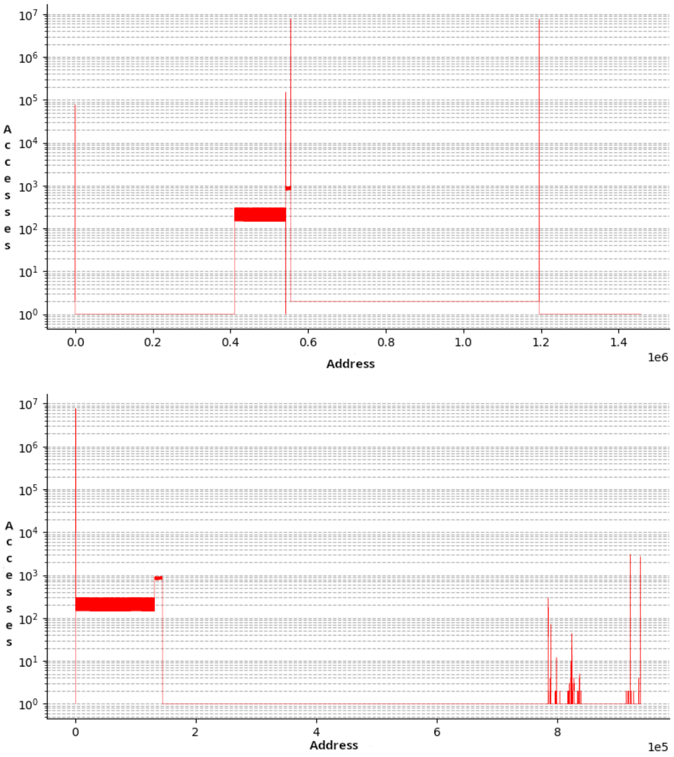


Fig. 6. Comparison of access counts from gem5 (full-system) simulation (top) and Valgrind (bottom), for the dijkstra benchmark. The x-axis shows normalized bit addresses and the y-axis shows the number of accesses.

Since usual applications for embedded systems are rather small, a recorded memory trace over several minutes should be sufficient to capture the representative access pattern.

B. Pre-Analysis and Indicator

The pre-analysis is performed on the benchmarks, compiled for and analysed on an AMD64 desktop workstation. This provides a fast analysis, compared to the simulation setup. However, memory is linked differently and addresses are virtual, as the benchmarks are measured on Linux and simulations are run on AARCH64 bare-metal. Therefore, the organization and ordering of memory is different for the pre-analysis. Furthermore, Valgrind does not trace the written memory content, but rather only provides a histogram of memory accesses. Thus, the pre-analysis only provides logic memory accesses and not the real amount of bit flips. However, the goal of the pre-analysis is to determine a ratio between the intensively used memory portions and occasionally used memory portions.

Figure 6 illustrates a comparison for the dijkstra benchmark between the real amount of memory accesses from our full-system simulation and the logic memory accesses from the pre-analysis. Indeed the memory layout of two analyses are different, but the trend of memory accesses for the first half of the memory space is comparable. For the example in Figure 7, we can observe that the memory accesses between the real amount and the logic memory accesses are drastically different than the example in Figure 6.

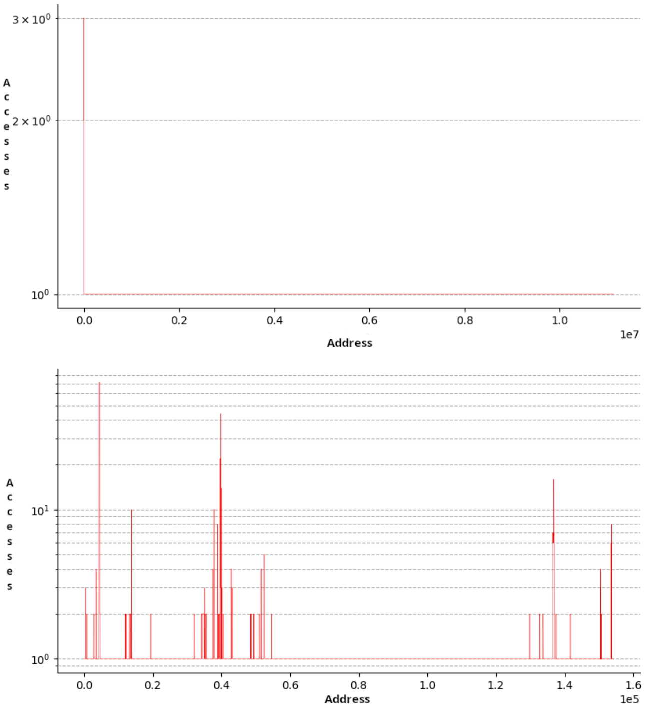


Fig. 7. Comparison of access counts from gem5 (full-system) simulation (top) and Valgrind (bottom), for the crc32 benchmark. The x-axis shows normalized bit addresses and the y-axis shows the number of accesses.

In order to quantify the results of the pre-analysis, we propose the **Pseudo Endurance** PE, similar to AE defined in Section III. The difference between these two metrics is that PE is calculated via access counts, gathered by Lackey. Similar to AE, a small value for PE should indicate that wear-levling methods should provide a note worthy life time improvement.

$$PE^{I}_{p(i)} = \frac{mean(\text{access\_count})}{max(\text{access\_count})} \tag{5}$$

For the example in Figure 6, the PE is reported as 0.00013 while the AE of the full-system simulation is reported as $3e^{-6}$. Figure 7 illustrates the same comparison for the crc32 benchmark, where the PE is reported as 0.016 and the AE as 0.99. It can be observed, that although the pseudo endurance differs largely from the achieved endurance, both indicators tend similarly to smaller and larger values for different benchmarks.

VI. Evaluation

In order to evaluate the performance of our approach, we conducted full-system simulations in gem5 with a cycle-accurate memory simulator and present the performance of the baseline and **Memory Carousel**, according to the four metrics defined in Section III. In addition, we present the results derived by our profiling tool to demonstrate the effectiveness of the pre-analysis for guiding the usage of **Memory Carousel**.

A. Evaluation Setup and Benchmarks

The evaluation setup was based on the software-managed wear leveling for NVMs by Hakert et al. [13]. All simulations were executed on a high-end AMD64 server. The programs were running on Unikraft [19], a library-based unikernel operating systems, and were simulated in gem5 with NVMain 2.0 [20], i.e., a cycle-accurate Non-Volatile-Main memory simulator. Within this setup, gem5 simulated a realistic **ARMv8** CPU. Although our solution should work on multi core systems, with minor changes to the synchronisation process, we decided to simulate a single core CPU. Since the simulation did not contain an operating system, Unikraft served as a runtime system and executed bare-metal on the system. Unikraft provides the required machine specific bootcode and drivers, but also basic primitives for memory management and rudimentary library support. NVMain, as a plugin extension to gem5, hooks into the simulation loop and is called on every single memory access. In this paper, we used the default memory trace configuration for NVMain, which has the purpose to only generate memory traces, since memory timing is not evaluated in this work. We extended NVMain with a custom trace writer, which provides detailed information about the bitwise wear-out.

The benchmarks consist of crc32, dijkstra, lesolve, quick-sort 64 bit (qsort-b), quick-sort 8 bit (qsort) and sha implementations. All these benchmarks were executed on the setup with two different implementations: 1) the original program, which is further called 'base', without any of our methods applied. 2) the program with our LLVM-IR pass and the trap triggered memory rotation, which is further called 'rot'. We also executed the program with our proposed LLVM-IR pass, patching all loads and stores, but without the memory rotation trap. A unpatched run is needed to arrive at a baseline of write accesses during the programs execution. This enables us to apply exactly 63 bit rotations on the memory word as mention in Section IV-D. All metrics used for analysis are described in Section III-A.

It is worth noting that our LLVM-IR pass was only applied to the benchmark programs C/C++ files and all operating system routines were not patched. Therefore, overhead introduced by the operating system was the same for all benchmarks, the rotation interrupt being the only exception. Also note that all presented results do not include the program stack.

B. Simulation and Analysis Results

In this section, we only focus on the results on the `data`, `bss` and `heap` memory interval I. For each benchmark we calculate our metrics, as presented in Section III. Please refer to Table I for the numbers. The first row of each benchmark is the "base" run, without our method applied. The second row shows the metrics of the "rot" run and additional metrics, comparing with the benchmarks base run. In context of our formalism presented in Section III the benchmark, e.g. crc32, would be the program p and "base" or "rot" the corresponding implementation i, so the AE of crc32's base run would map to $AE^{I}_{crc32(base)}$. All metrics are multiples of the base run and are therefore unitless. With the Achieved Endurance AE being the only exception, this is comparing against a theoretical ideal memory distribution with even wear-out. For better comparison the Lifetime Improvement of all six benchmarks is also visualized in Fig. 8.

TABLE I
SIMULATION RESULTS FOR DATA, HEAP AND BSS ON OUR BENCHMARKS.

	Achieved Endurance AE	Endurance Improvement EI	Overhead OV	**Lifetime Improvement** LI
crc32				
-base	0.99×			
-rot	0.34×	0.35×	15.62×	0.02×
dijkstra				
-base	$3.2e^{-6}\times$			
-rot	$66e^{-6}\times$	20.80×	1.13×	18.44×
lesolve				
-base	$4e^{-3}\times$			
-rot	$5e^{-3}\times$	1.42×	1.17×	1.21×
qsort				
-base	$1e^{-4}\times$			
-rot	$126e^{-4}\times$	95.34×	4.41×	21.61×
qsort-b				
-base	$4e^{-5}\times$			
-rot	$168e^{-5}\times$	42.84×	2.78×	15.40×
sha				
-base	0.86×			
-rot	0.66×	0.76×	32.00×	0.02×
lesolve+				
-opt-rot	$3e^{-3}\times$	0.93×	0.03×	34.41×

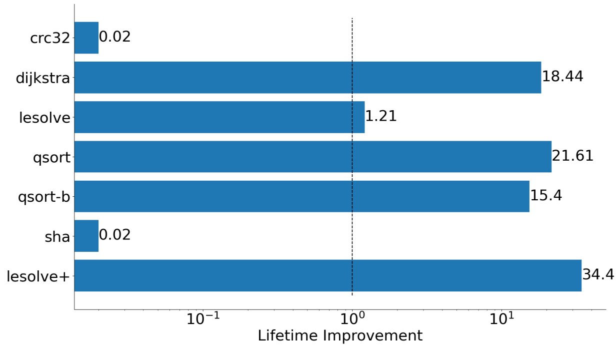


Fig. 8. Half logarithmic diagram of the Lifetime Improvements for all six Benchmarks.

In addition, we simulated a block wear-leveling approach as described in Section IV-E. Assessing the overhead of such an approach, would require to track the memory content of an entire block in order to determine the amount of bitflips upon the relocation of a block. Since this is not feasible for our full-system simulation, we are limited to a best case assumption, i.e. that no bit flips are caused upon a relocation and a worst case assumption, i.e. that all bits are flipped upon a rotation. This leads to generating four results for each benchmark with an optimistic and pessimistic result for each of the two runs, thus a lower and a upper bound. Since neither the optimistic, nor the pessimistic result is realistic, we focus our discussion on the Endurance Improvement EI for these results, please see Fig. 9. For completeness all other metrics can be found in Table II.

At a first glance it can be seen that, our approach worsens the memory lifetime for two benchmarks, namely: crc32 and sha. However both crc32 and sha do not write to the targeted

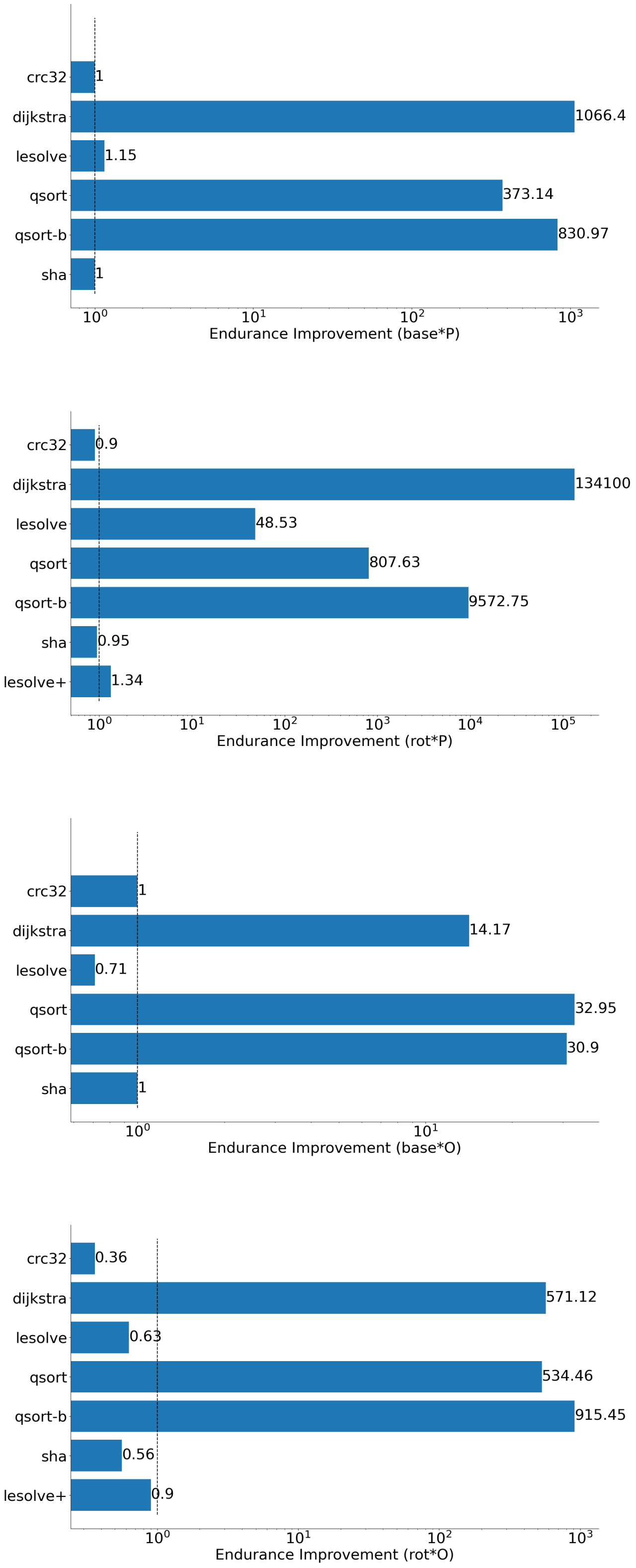


Fig. 9. Half logarithmic diagrams of the Endurance Improvement‘s EI of the simulated memory block wear levelling.

TABLE II
SIMULATION RESULTS WITH ADDITIONAL BLOCK WEAR-LEVELLING SIMULATION. WHERE THE SUFFIX *P REPRESENTS THE PESSIMISTIC AND *O THE OPTIMISTIC SIMULATION.

	Achieved Endurance AE	**Endurance Improvement** EI	Overhead OV	Lifetime Improvement LI
crc32				
-base*P	0.99×	1.00×	1.00×	1.00×
-rot*P	0.89×	0.90×	101365.77×	$9e^{-6}$×
-base*O	0.99×	1.00×	1.00×	1.00×
-rot*O	0.36×	0.36×	15.62×	0.02×
dijkstra				
-base*P	$3e^{-3}$×	1066.40×	38.30×	27.84×
-rot*P	$424e^{-3}$×	134100.03×	11159.20×	12.02×
-base*O	$0.05e^{-3}$×	14.17×	0.46×	30.89×
-rot*O	$2e^{-3}$×	647.21×	1.13×	573.99×
lesolve				
-base*P	$4e^{-3}$×	1.15×	7.03×	0.16×
-rot*P	$171e^{-3}$×	48.53×	6169.12×	0.01×
-base*O	$3e^{-3}$×	0.71×	0.69×	1.03×
-rot*O	$5e^{-3}$×	1.46×	1.17×	1.25×
qsort				
-base*P	$49e^{-3}$×	373.14×	140.36×	2.66×
-rot*P	$107e^{-3}$×	807.63×	2036.56×	0.40×
-base*O	$4e^{-3}$×	32.95×	0.97×	33.91×
-rot*O	$80e^{-3}$×	603.83×	4.41×	136.84×
qsort-b				
-base*P	$33e^{-3}$×	830.97×	409.58×	2.03×
-rot*P	$377e^{-3}$×	9572.75×	43305.68×	0.22×
-base*O	$1e^{-3}$×	30.90×	0.97×	31.94×
-rot*O	$42e^{-3}$×	1074.48×	2.78×	386.10×
sha				
-base*P	0.86×	1.00×	1.00×	1.00×
-rot*P	0.82×	0.95×	217340.41×	$4.4e^{-6}$×
-base*O	0.86×	1.00×	1.00×	1.00×
-rot*O	0.56×	0.65×	32.00×	0.02×
lesolve+				
-opt-rot*P	$5e^{-3}$×	1.34×	169.61×	0.01×
-opt-rot*O	$2e^{-3}$×	0.61×	0.02×	32.08×

memory, resulting in an AE of nearly 1. The simulated block wear levelling does not introduce an overhead in contrast to our method. This is the case since block wear levelling, as simulated, only triggers after a specific amount of writes. In contrast, our method is applied in a way it triggers exactly 63 times over the benchmark duration and therefore introducing an overhead to unwritten memory. Also worth mentioning is that all block wear-levelling simulations improve the worsened EI of our approach for those benchmarks. Due to the existence of such anomalies we propose a profiling method as described in Section V.

Also noteworthy is lesolve, where our approach is only able to slightly improve upon. The reason for this is, that lesolve works on a very small memory region on `data` and the remaining memory behaves similar to crc32 and sha. However, as discussed in Section IV-C, our method can be applied to an arbitrary memory interval, inside `data` , `bss` and `heap` , and therefore it can be optimized for such cases.

To show this, we measure an optimized version of lesolve (opt-rot), where we apply our wear-levelling solution only to a small memory interval. This interval is the memory region with the most accesses in the benchmark. After this optimization lesolve opt-rot becomes the best performing benchmark of our method with a lifetime improvement LI of 34.41×. Please note that this result uses a different memory interval and should not be directly compared to other results in the tables and figures. However, such optimization requires a deep understanding of the program memory access patterns and can only be utilized after time consuming analyses like presented here. Moreover, all accesses occur only within one array, holding intermediate results. If there is a need to wear level multiple data objects, they have to be linked in a single interval for our method to still be applicable.

In summary, in three of six benchmarks, **Memory Carousel** provides a significant lifetime improvement LI, namely dijkstra, qsort and qsort-b. They obtain an $LI > 15$ and their EI's are further improved by the block wear-leveling simulation. However, for the optimized lesolve opt-rot, the block wear leveling does not provide a significant gain.

TABLE III
MULTIPLIER OF MEMORY AND CPU CYCLES OF THE PATCHED EXECUTION COMPARED TO NORMAL EXECUTION.

	crc32	dijkstra	lesolve	qsort	qsort-b	sha
Memory						
with rot	12.1×	14.0×	10.7×	4.1×	6.4×	10.4×
patched only	11.2×	14.0×	10.7×	4.1×	6.4×	10.3×
CPU						
with rot	12.1×	14.0×	10.7×	4.2×	6.4×	10.4×
patched only	11.2×	14.0×	10.7×	4.1×	6.4×	10.3×

Although many concepts towards wear-leveling on different garnularities exist in related work, a direct comparison is usually challenging. Not only is source code rarely published and straight forward applicable, but also the assumptions about the memory and wear-out model differ a lot. A war-leveling scheme, designed for non iterative write scheme memories may achieve totally different results on a memory with iterative write semantic. Nonetheless, we intend to give a rough intuition to the range of lifetime improvement, other published work can achieve. The work, we base our simulation system on [13], provides an MMU based coarse-grained wear-leveling methods, which can achieve a lifetime improvement (considering caused overheads) of 10× to 30×. A fine-grained extension, targeting the stack memory can further achieve an improvement of a few hundred times for a small set of specific benchmarks. Another page based wear-leveling scheme [21] also reports lifetime improvements in a range of 5× to 200× for various approaches and benchmarks. Hence, the lifetime improvement of memory carousel plays in a similar range as other published methods and is intended to be a compatible extension towards such methods.

C. Execution Time Overhead

Our approach does not only affect the memory wear-out, it also increases the execution time of the benchmarks. To measure the increase in execution time, we took the simulated execution cycles of the memory controller and CPU in gem5. We decided to use the memory controller cycles because they are easily accessible with the NVMain module from gem5. Also the difference to CPU cycles is minimal. The measured cycles of the rotated run are divided by the cycles of the base run, to provide the multiple of the cycles our approach needs compared to normal execution.

TABLE IV
PSEUDO ENDURANCE PE MEASURED WITH VALGRIND FOR ALL BENCHMARKS.

	crc32	dijkstra	lesolve	qsort	qsort-b	sha
PE	0.01592	0.00013	0.01161	0.00033	0.00055	0.01626

As shown in the first row of Table III, dijkstra needs 14 times longer with our approach applied and has the highest cycle multiplier of all benchmarks. The qsort benchmark has the smallest multiplier of 4.1. The remaining 4 benchmarks are nearly evenly distributed between these two. So the run-time impact depends strongly on the program. The second row shows the cycle multiplier of the patched programs without the memory rotation trap. Since the values in both rows do not differ much from each other, we can say that the actual rotation of the memory does not contribute much to the run-time overhead. Most overhead is introduced by conditional branching before every load and store operation. Moreover, the number of rotation is exactly ×63 and thus constant. In contrast, the additional branches behave in a linear manner to the run-time, so the longer the program executes the more branches are executed, while the memory rotation from the operating system service stays the same. Overall, the longer the benchmark runs, the smaller the impact of memory rotation.

D. Results of Pre-Analysis

All calculated PE's on $I =$ [Data, BSS and Heap] can be found in Table IV. As presented previously, indeed our method is not able to improve the lifetime of crc32 and sha, but does so for the remaining four benchmarks. This can be observed to be clearly reflected in the pseudo endurance, i.e., the result is larger by two orders of magnitude for the benchmarks, which cannot be improved by our method. Except for lesolve, which only provided a diminishing lifetime improvement. This suggests that the Valgrind based profiling can be well used to estimate in advance whether the overheads of our method can be leveraged by the gained lifetime improvement. Although we could simply define a decisive threshold for the pseudo endurance in our scenario, this would be dependant on the system and also on the configuration, i.e. how often the rotation is supposed to happen.

E. Comparison to State-of-the-Art

In the literature, several approaches for NVM wear-leveling can be found. Table V shall provide a brief comparison among these and our method, whereas Section VII provides more information about each approach. We compare the granularity of memory units for wear-leveling, the achieved lifetime improvement, whether a method moves entire blocks or bits within a block, if the method is aging aware, requires special hardware or a general MMU, and if it is applicable to general applications or only special software. One method can switch

[1] This work only provides a relative comparison to other approaches, but no absolute numbers.

TABLE V
COMPARISON OF STATE-OF-THE-ART NVM WEAR-LEVELING METHODS

	WL Granularity	Lifetime Improvement	Block Based	Aging Aware	Special Hardware needed	MMU independent	General Applicable
Memory Carousel	64 bit	21×	✗	✗	✗	✓	✓
Enhanced Wear-Rate Leveling [7]	4 MB	17×	✓	✓	✓	✗	✓
Kevlar [22]	4kB	31.7×	✓	✓	✗	✗	✓
WoLFRaM [23]	256B-1kB	N/A[1]	✓	✓	✓	✗	✓
Increasing PCM Main Memory Lifetime [24]	2kB	28.91×	✓	✓/ ✗	✓	✗	✓
Software-Managed Read and Write Wear-Leveling [25]	4kB	955×	✓	✓	✗	✗	✓
Lewat (KV Allocation) [10]	256B	5000×	✓	✓	✗	✗	✗
Flip-N-Write [4]	2-32 bit	2.7×	✗	✗	✓	✓	✓
Balanced Gray Codes [6]	8 bit	2×	✗	✗	✓/ ✗	✓	✗

whether or not it is aging aware. Another method can switch whether or not it is implemented on special hardware.

It should be noted that most are block based, i.e., they remap memory blocks as a wear-leveling action and therefore they do not operate on a bit granularity, like our method. It should be further noted, that some methods either propose special hardware or rely on the availability of an MMU. Flip-N-Write [4] operates on bit granularity, but requires special hardware. Balanced Gray Codes [6], in contrast, are only applicable to numeric values, i.e., they are not generally applicable.

Although many state-of-the-art approaches achieve a significantly higher lifetime improvement in comparison to Memory Carousel, none of them can operate under the same system assumptions of extremely limited hardware. Hence, Memory Carousel can still help to improve the lifetime of NVM systems, when other methods are not applicable.

VII. RELATED WORK

In the literature, several previous works have been proposed against the limited endurance of non-volatile memories, which is highly related to the lifetime. They range from working on fine-grained levels [4]–[6], [11] to coarse-grained memory blocks [7]–[9] or even with multiple granularities like [10], [13]. To improve memory lifetime, the previous works used either aging-aware strategies, e.g., [16], [17], [21]–[23], or non-aging-aware strategies, e.g., [11], [15], [24]. For the completeness, we select some representatives to review their insights and thus position our work in the following.

The principle of aging-aware strategies is to assess the age of cell via tracking memory accesses to apply wear-leveling. For example, Han et al. proposed to predict the next possible writings and swaps the areas where the data may be written [7]. Gogte et al. adopted a sampled-based approach to approximate the write distribution, together with an advanced debugging feature offered by Intel [22]. On the contrary, the non-aging-aware strategies do not track the access patterns but rather perform their actions periodically [9], [15] or randomly, e.g., [5], [12], [24]. Zhou et al. proposed to shift the data of a memory row one byte at a time periodically [15]. Curling-PCM periodically moves the hot areas over the memory [9]. It can be configured to manage the memory space in different granularity. Qureshi et al. proposed to randomize the address-space together with the well-known Start-Gap approach, which keeps moving one memory line from its location to a neighboring location. [12]. Another example is Walloc that uses lazy copy over write and scatters the data all over the free memory in a ”Less Allocated First Out” manner [5].

A main challenge with most of these wear-leveling solutions is that they need modification or special supports of the underlying hardware which cannot be trivially integrated with other systems. Alternatively, software-based approaches, which are relatively more portable, have been more attractive. WoLFRAM uses a programmable resistive address decoder to change the address and swap it in a write-access-pattern aware manner, by adding one specific controller for each memory bank [23]. Hakert et al. proposed to use a red-black tree to maintain the estimated age of physical memory pages without special hardware supports [14]. Huang et al. proposed to change the file system structure into a new structure to provide different granularity levels for reducing the NVM wear [10]. However, none of them have tackled the wear-out of memories employing the iterative write scheme.

A few existing solutions in the literature are similar to ours. Flip-N-Write uses one extra bit to either store the data in the right order or reversed [4], which relies on the modification of microarchitecture. The decision is done based on comparing the to-be-written data with the data that is already stored in the same location. The format that requires less bit flips should be applied. Zhao et al. proposed a strategy to flip data within the memory based on a write count in order to level out how much data is written to the same location [11]. However, the realization details are not discussed. Recently, Kulandai et al. proposed to use gray coding of the data in order to achieve the least possible bit flipping between different writes [6]. However, as stated by the authors, additional hardware support is needed to translate nibbles and bytes from the integer representation to Gray codes, which might not be realistic. Overall, Memory Carousel differs than all of them in the fact that it does not require any hardware modification.

VIII. Conclusion

In this paper, we present **Memory Carousel**, a software based solution to the problem of wear-leveling iterative write scheme non-volatile memories. Within this method, we continuously rotate the applications memory in order to spread the wear-out of intensively flipped bits evenly across memory words. We realize applications correctness with a LLVM pass, patching all load and store operations. The major drawback of this approach is that the application stack cannot be wear-leveled due to spill and fill operations and the calling conventions. Our method could be extended to also wear-level the stack, when patching assembly code directly or LLVM machine intermediate representation, instead of LLVM IR. This, however, would make the solution dependant on the system architecture, which is also considered out of scope.

Extensive evaluation highlights that software based bit wear-leveling has to be carefully applied. For certain benchmark applications, our method causes an overhead, which exceeds the gained improvement by far. On the other hand, when applied to a different subset of applications, we can achieve a significant lifetime improvement of up to 21× and even allow further potential for coarser grained wear-leveling. Nevertheless, with the help of our valgrind based offline profiling, we can clearly separate the applications, which gain lifetime improvement from the ones, which cause unreasonable overheads. Thus, when profiling a target application upfront, we can apply our method to meaningful scenarios only. We further highlight that if not the entire memory space is wear-leveled, but the worn-out regions are chosen carefully, further lifetime improvement of up to 34× can be achieved on an application that would slightly profit otherwise. This however is only possible with a deep understanding of the programs access pattern and worn-out memory regions have to be identified by the programmer, as currently no automated analysis exists.

For future work, we identify the choice of memory regions of interest as a crucial problem. When limiting the wear-leveling to such regions only, higher lifetime improvements can be gained. In consequence, we aim to extend our valgrind based profiling tool to already identify such regions upfront and configure the wear-leveling accordingly.

Acknowledgement

This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG), as part of the Collaborative Research Center under Grant SFB876 (124020371, http://sfb876.tu-dortmund.de), subproject A1, OneMemory (405422836), and ARTS-NVM (502308721).

References

[1] M. N. I. Khan, A. Jones, R. Jha, and S. Ghosh, *Sensing of Phase-Change Memory*. Springer International Publishing, 2019, pp. 81–102.

[2] M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras, “Preset: Improving performance of phase change memories by exploiting asymmetry in write times,” *SIGARCH Computer Architecture News*.

[3] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A durable and energy efficient main memory using phase change memory technology,” *ACM SIGARCH computer architecture news*, vol. 37, no. 3, pp. 14–23, 2009.

[4] S. Cho and H. Lee, “Flip-N-write: A simple deterministic technique to improve PRAM write performance, energy and endurance,” *International Symposium on Microarchitecture (MICRO)*, pp. 347–357, 2009.

[5] S. Yu, N. Xiao, M. Deng, Y. Xing, F. Liu, Z. Cai, and W. Chen, “WAlloc: An efficient wear-aware allocator for non-volatile main memory,” *34th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, 2015.

[6] A. D. R. Kulandai, S. J, J. Rose, and T. Schwarz, *Balanced Gray Codes for Reduction of Bit-Flips in Phase Change Memories*. Springer International Publishing, 2021, vol. 12527 LNCS. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-68110-4_11

[7] Y. Han, J. Dong, K. Weng, Y. Wang, and X. Li, “Enhanced Wear-Rate Leveling for PRAM Lifetime Improvement Considering Process Variation,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 1, pp. 92–102, 2016.

[8] X. Chen, E. H. Sha, X. Wang, C. Yang, W. Jiang, and Q. Zhuge, “Contour: A Process Variation Aware Wear-Leveling Mechanism for Inodes of Persistent Memory File Systems,” *IEEE Transactions on Computers*, vol. 70, no. 7, pp. 1034–1045, 2021.

[9] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha, “Curling-PCM: Application-specific wear leveling for phase change memory based embedded systems,” *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pp. 279–284, 2013.

[10] K. Huang, S. Li, L. Huang, K. L. Tan, and H. Mei, “Lewat: A Lightweight, Efficient, and Wear-Aware Transactional Persistent Memory System,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 649–664, 2021.

[11] M. Zhao, L. Shi, C. Yang, and C. J. Xue, “Leveling to the last mile: Near-zero-cost bit level wear leveling for PCM-based main memory,” *32nd IEEE International Conference on Computer Design (ICCD)*, pp. 16–21, 2014.

[12] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, “Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling,” in *International Symposium on Microarchitecture (MICRO)*, 2009, pp. 14–23.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 17, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[14] C. Hakert, K.-H. Chen, M. Yayla, G. von der Brüggen, S. Blömeke, and J.-J. Chen, “Software-based memory analysis environments for in-memory wear-leveling,” in *IEEE 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020, pp. 651–658.

[15] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A durable and energy efficient main memory using phase change memory technology,” *Proceedings - International Symposium on Computer Architecture*, pp. 14–23, 2009.

[16] W. Zhang and T. Li, “Characterizing and mitigating the impact of process variations on phase change based memory systems,” *International Symposium on Microarchitecture (MICRO)*, pp. 2–13, 2009.

[17] C. H. Chen, P. C. Hsiu, T. W. Kuo, C. L. Yang, and C. Y. M. Wang, “Age-based PCM wear leveling with nearly zero search cost,” *Proceedings - Design Automation Conference*, pp. 453–458, 2012.

[18] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” ser. PLDI ’07. New York, NY, USA: ACM, 2007, p. 89100.

[19] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici, “Unikraft: Fast, specialized unikernels the easy way,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 376394.

[20] M. Poremba, T. Zhang, and Y. Xie, “Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems,” *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, 2015.

[21] H. A. Khouzani, Y. Xue, C. Yang, and A. Pandurangi, “Prolonging PCM lifetime through energy-efficient, segment-aware, and wear-resistant page allocation,” *Proceedings of the International Symposium on Low Power Electronics and Design*, vol. 2015-October, pp. 327–330, 2015.

[22] V. Gogte, P. M. Chen, S. Narayanasamy, T. F. Wenisch, W. Wang, S. Diestelhorst, and A. Kolli, “Software wear management for persistent memories,” *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*.

[23] L. Yavits, L. Orosa, S. Mahar, J. D. Ferreira, M. Erez, R. Ginosar, and O. Mutlu, “WoLFRaM: Enhancing Wear-Leveling and Fault Tolerance in Resistive Memories using Programmable Address Decoders,” *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 187–196, 2020.

[24] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing PCM main memory lifetime," *Proceedings - Design, Automation and Test in Europe, DATE*, pp. 914–919, 2010.

[25] C. Hakert, K.-H. Chen, H. Schirmeier, L. Bauer, P. R. Genssler, G. von der Brüggen, H. Amrouch, J. Henkel, and J.-J. Chen, "Software-managed read and write wear-leveling for non-volatile main memory," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 1, feb 2022. [Online]. Available: https://doi.org/10.1145/3483839

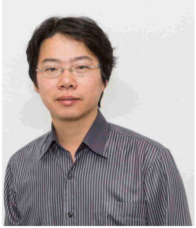
Nils Hölscher is a research associate at TU Dortmund in the group of Design Automation for Embedded Systems with Prof. Jian-Jia Chen. He received his Master degree in Computer Science from TU Dortmund in 2021. His research interest is the support and application of compiler solutions for embedded systems for both non-volatile memory and real-time properties.

Christian Hakert is a research associate at TU Dortmund in the group of Design Automation for Embedded Systems with Prof. Jian-Jia Chen. He received his Master degree in Computer Science from TU Dortmund in 2019 and received the best student award ”Jahrgangsbestenpreis” for his master degree in 2019. His research interest is the support and application of non-volatile main memories in system software and operating systems.

Hassan Nassar received his M.Sc degree from Ulm University, Germany in 2019 and his B.Sc degree –with highest honours– from the German University in Cairo, Egypt, in 2016. He joined the Chair for Embedded Systems in March 2020 as a research assistant. His research interests are hardware security, Reconfigurable Architectures, Memory Reliability, and Cloud FPGAs.

Kuan-Hsun Chen is a tenured assistant professor in the Department of Computer Science at University of Twente in the Netherlands. From Aug. 2019 to Aug. 2021, he was a postdoc at TU Dortmund University in Germany. He earned his Ph.D. (Dr.-Ing.) in Computer Science from TU Dortmund University with a distinction Summa cum Laude in 2019. He earned his master's degree in Computer Science from National Tsing Hua University (Taiwan) in 2013. His research interests include real-time embedded systems, architecture-aware software design, and dependable computing.

Lars Bauer received the M.Sc. and Ph.D. degrees in computer science from the University of Karlsruhe, Germany, in 2004 and 2009. He is currently a research group leader and lecturer at the Chair for Embedded Systems (CES) at the Karlsruhe Institute of Technology (KIT). Dr. Bauer received two dissertation awards (EDAA and FZI), two best paper awards (AHS'11 and DATE'08) and several nominations. His research interests include architectures and management for adaptive multi-/manycore systems.



Jian-Jia Chen is Professor at Department of Informatics in TU Dortmund University in Germany. He was Juniorprofessor at Department of Informatics in Karlsruhe Institute of Technology (KIT) in Germany from May 2010 to March 2014. He received his Ph.D. degree from Department of Computer Science and Information Engineering, National Taiwan University, Taiwan in 2006. He received his B.S. degree from the Department of Chemistry at National Taiwan University 2001. Between Jan. 2008 and April 2010, he was a postdoc researcher at ETH Zurich, Switzerland. His research interests include real-time systems, embedded systems, energy-efficient scheduling, power-aware designs, temperature-aware scheduling, and distributed computing. He received the European Research Council (ERC) Consolidator Award in 2019. He has received more than 10 Best Paper Awards and Outstanding Paper Awards and has involved in Technical Committees in many international conferences.

Jörg Henkel received the Diploma and Ph.D. (summa cum laude) degree from the Technical University of Braunschweig. He is currently the Chair Professor of embedded systems with the Karlsruhe Institute of Technology. Before that he was a Research Staff Member with NEC Laboratories, Princeton, NJ, USA. His research interest includes co-design for embedded hardware/software systems with respect to power security and means of embedded machine learning. He has led several conferences as a General Chair including ICCAD and ESWeek, and is currently DAC Vice Chair. He serves as a steering committee chair/member for leading conferences and journals for embedded and cyber-physical systems. He has coordinated the DFG Program SPP 1500 "Dependable Embedded Systems" and is a Site Coordinator of the DFG TR89 Collaborative Research Center on "Invasive Computing". He is the Chairman of the IEEE Computer Society, Germany Chapter. He has received six best paper awards throughout his career from, among others, ICCAD, ESWeek, and DATE. For two consecutive terms each, he served as the Editor-in-Chief for both the ACM Transactions on Embedded Computing Systems and the IEEE Design & Test magazine. He is the Vice President for Publications at IEEE CEDA and a Fellow of the IEEE.