Software-Managed Read and Write Wear-Leveling for Non-Volatile Main Memory

CHRISTIAN HAKERT, KUAN-HSUN CHEN, and HORST SCHIRMEIER, Technical University of Dortmund
LARS BAUER, Karlsruhe Institute of Technology
PAUL R. GENSSLER, University of Stuttgart
GEORG VON DER BRÜGGEN, Max Planck Institute for Software Systems
HUSSAM AMROUCH, University of Stuttgart
JÖRG HENKEL, Karlsruhe Institute of Technology
JIAN-JIA CHEN, Technical University of Dortmund

In-memory wear-leveling has become an important research field for emerging non-volatile main memories over the past years. Many approaches in the literature perform wear-leveling by making use of special hardware. Since most non-volatile memories only wear out from write accesses, the proposed approaches in the literature also usually try to spread write accesses widely over the entire memory space. Some non-volatile memories, however, also wear out from read accesses, because every read causes a consecutive write access. Software-based solutions only operate from the application or kernel level, where read and write accesses are realized with different instructions and semantics. Therefore different mechanisms are required to handle reads and writes on the software level. First, we design a method to approximate read and write accesses to the memory to allow aging aware coarse-grained wear-leveling in the absence of special hardware, providing the age information. Second, we provide specific solutions to resolve *access hot-spots* within the compiled program code (text segment) and on the application stack. In our evaluation, we estimate the cell age by counting the total amount of accesses per cell. The results show that employing all our methods improves the memory lifetime by up to a factor of 955×.

CCS Concepts: • **Computer systems organization → Embedded systems**; • **Hardware → Memory and dense storage;**

Additional Key Words and Phrases: Non-volatile memory, wear-leveling, read-destructive, age approximation

5

This work was supported by Deutsche Forschungsgemeinschaft (DFG), as part of the project OneMemory (no. 405422836) and the project SFB876 A1 (no. 124020371).

Authors' addresses: C. Hakert, Otto-Hahn Straße 16, Raum E20, 44227 Dortmund, Germany; email: christian.hakert@tu-dortmund.de; K.-H. Chen, University of Twente P.O. Box 217, 7500 AE Enschede, The Netherlands; email: k.h.chen@utwente.nl; H. Schirmeier, Andreas-Pfitzmann-Bau, Raum 3102, Nöthnitzer Straße 46, 01187, Dresden, Germany; email: horst.schirmeier@tu-dresden.de; L. Bauer, Haid-und-Neu-Str. 7, Bldg. 07.21, 76131 Karlsruhe, Germany; email: lars.bauer@kit.edu; P. R. Genssler, Pfaffenwaldring 47, D-70569 Stuttgart, Room: 3.170, Germany; email: Paul.Genssler@informatik.uni-stuttgart.de; G. von der Brüggen, Otto-Hahn Straße 16, Raum E19, 44227 Dortmund, Germany; email: georg.von-der-brueggen@tu-dortmund.de; H. Amrouch, Pfaffenwaldring 47, D-70569 Stuttgart Room: 3.163, Germany; email: amrouch@iti.uni-stuttgart.de; J. Henkel, Haid-und-Neu-Str. 7, Bldg. 07.21, 76131 Karlsruhe, Germany; email: henkel@kit.edu; J.-J. Chen, Otto-Hahn Straße 16, Raum E21, 44227 Dortmund, Germany; email: jian-jia.chen@cs.tu-dortmund.de.

1539-9087/2022/02-ART5
https://doi.org/10.1145/3483839

ACM Reference format:
Christian Hakert, Kuan-Hsun Chen, Horst Schirmeier, Lars Bauer, Paul R. Genssler, Georg von der Brüggen, Hussam Amrouch, Jörg Henkel, and Jian-Jia Chen. 2022. Software-Managed Read and Write Wear-Leveling for Non-Volatile Main Memory. *ACM Trans. Embedd. Comput. Syst.* 21, 1, Article 5 (February 2022), 24 pages. https://doi.org/10.1145/3483839

1 INTRODUCTION

In recent years, **non-volatile memory (NVM)** has been considered as an alternative to DRAM or SRAM for memory. Due to several drawbacks (e.g., a lower cell endurance), maintenance strategies have been proposed in the literature to overcome the impacts. Memory lifetime is a crucial issue when NVMs are considered, because it can shrink to hours or even minutes when no maintenance is applied [5, 9]. Therefore, wear-leveling strategies try to extend the memory lifetime to the maximum by stressing all memory cells equally over time. Wear-leveling strategies can be categorized as aging-aware and non-aging-aware strategies, where aging-aware strategies investigate the current cell ages to make adequate wear-leveling decisions. This requires precise knowledge about the cell ages, which can be either gathered from special hardware or can be approximated by software. Non-aging-aware strategies, in contrast, base their wear-leveling decisions on other mechanisms (e.g., on random). Most aging-aware and non-aging-aware strategies usually only target write accesses, because only write accesses wear out the memory cells for most NVM types.

Some NVMs, such as **Ferroelectric RAM (FeRAM)**, are also read-destructive [18]. This means that every read access also wears out the memory cell. For FeRAM, the reason is that read accesses overwrite the current cell value and thus it has to be recharged subsequently [18]. Therefore, every read access results in an automatic subsequent write access to the cell. This process is triggered by the memory controller during every read access. Consequently, read accesses have to be considered with the same impact as write accesses during wear-leveling if such read-destructive memories are used. Although this does not imply a significant difference on the hardware level (especially hardware-based wear-leveling), because hardware can simply be extended to track read accesses as part of the wear-leveling, it does so on the software level. If wear-leveling is realized in software, tracking and counting read accesses differs from write accesses. Read and write accesses happen with different semantics on the software level, and indeed there is various "read-only" data in most programs. Thus, the different semantics have to be considered during the design of software-based solutions.

In this article, we study the design of software-managed wear-leveling, where we investigate the design and usage of age approximations for aging-aware wear-leveling in the absence of special hardware. By our software-managed mechanism, we hereby denote a solution that is not fully independent of the underlying hardware and still requires a certain set of system features. We rather consider solutions, which configure commonly available hardware components from the software level and therefore can be implemented purely on the software level of state-of-the-art systems. More specifically, our solution requires an MMU and performance counters, which can be found in many recent application processors. The solution, however, does not propose a modification of the aforementioned hardware but only manages and configures it from the software level to achieve the wear-leveling goals. A software-based solution in general must not be favorable over a hardware-based solution, but rather offers a wear-leveling solution for systems where the required hardware support is not implemented, either on purpose to save chip space or because the system hardware is already configured. We further cover the destructive influence of read and write accesses within our method, such that read-destructive NVMs can be targeted as well. First, we design coarse-grained aging-aware wear-leveling, where we sample read and write

accesses to the main memory with the help of performance counters and memory access permissions. This provides us a statistical approximation of the read and write access distribution. We feed this approximation into a virtual memory page based wear-leveling algorithm, which remaps the physical memory pages behind the virtual memory pages in an aging-aware manner.

As an orthogonal, we further design non-aging-aware *fine-grained* wear-leveling, which resolves dense access hot-spots within memory pages. The application stack segment faces intensive hot-spots of dense read and write accesses, whereas compiled program code (i.e., in the text segment) only faces read hot-spots. Targeting this, we propose a mechanism that moves the stack and the text segment in a circular manner with small offsets through the physical memory while the correct execution of the program is ensured. In combination with the preceding coarse-grained aging-aware wear-leveling, we achieve allover wear-leveled memory, even in the presence of a read-destructive main memory.

Our novel contributions include the following:

- An online read and write approximation to allow coarse-grained page swapping with respect to read and write accesses.
- A fine-grained wear-leveling for the stack segment to avoid write and read hot-spots. The stack is moved in a rotational manner while correctness of pointers is maintained.
- A fine-grained wear-leveling for the text segment to avoid read hot-spots. This approach moves the text segment in a rotational manner to spread read hot-spots through a larger memory area.

2 RELATED WORK

Over the past decades, several approaches for in-memory wear-leveling for NVM have been proposed. These approaches can be categorized along different criteria. First, there are aging-aware approaches [1, 2, 4–6, 11, 15, 17, 20, 24], which take the current cell age into account to apply wear-leveling. In contrast, randomized approaches [5, 19, 24] apply wear-leveling in a circular or randomized manner. Both approaches are often combined to achieve a randomized wear-leveling on fine granularities inside of memory blocks, whereas an aging-aware approach is used to target these coarse-grained memory blocks. The granularity also varies from single bits [3, 23] over cache-lines [19, 24] for fine-grained approaches to memory pages [1, 2, 5, 6, 20] or even bigger memory segments [22, 24] for coarse-grained approaches.

Some approaches are not based on remapping the physical memory content through an abstraction layer, but hook into the memory allocation process of the operating system to apply wear-leveling to the memory allocator [1, 15, 20]. Li et al. [15] also propose to use an allocated memory portion, whenever a function is called, for the function's stack memory to wear-level the stack region.

2.1 Aging-Aware Wear-Leveling

Gogte et al. [6] propose a software-only coarse-grained wear-leveling approach by using a sampled approximation of the write distribution. They make use of advanced debugging capabilities, such as Intel Processor Event Based Sampling (PEBS), which allows them to sample the write requests from the CPU. These debugging capabilities, however, can rarely be found in embedded systems and resource-constrained hardware.

All other mentioned aging-aware approaches rely on the current write-count information of the memory. Most approaches introduce specialized hardware into the memory controller to collect the write-count information, which is not available in commonly available systems and might be hard to realize. Dong et al. [4] use an offline recorded memory trace to estimate the write distribution, which limits the approach to a subset of well-known applications only.

2.2 Read Wear-Leveling

To the best of our knowledge, there are no dedicated algorithms for read wear-leveling in read-destructive NVMs. However, hardware-based approaches that are not aging-aware or that directly decide based on the wear of each cell are compatible with read-destructive memories by default. If the wear is estimated from the write count, it could be also estimated from the read and write count together. This implies that hardware-software interplay algorithms can obtain the accurate wear estimation by extending the hardware to count read accesses as well. As long as generic mechanisms (e.g., virtual memory page remapping) are used [1, 2], the modifications to the algorithms are minimal. In contrast, when specific mechanisms (e.g., heap allocation or stack allocation) are used for wear-leveling [14, 15, 20], then read wear-leveling cannot be integrated easily. Thus, another special mechanism for read wear-leveling is required in those cases. In addition, for algorithms that ship with their own write approximation [6, 9, 12], a specialized read approximation has to be added.

To the best of our knowledge, this is the first work to propose a software-managed memory read and write access approximation that does not rely on special debug capabilities. Furthermore, specific algorithms are provided, which operate on application-specific data (stack and text), where the algorithm for text is dedicated to read accesses only.

3 TARGET SYSTEM

In this section, we first scope the typical target setup of our proposed methods. Although we do not limit our methods to specific NVM types, we consider certain properties regarding the wear-out. We assume any cell modification wears out a cell equally. We further do not assume iterative writing, and thus the number of memory accesses corresponds to the memory lifetime linearly. In our proposed method, we target write destructive, as well as read-destructive, NVMs by implementing an add-on for read-destructive NVMs. Although the read-destructive property may not only be found in one NVM type, FeRAM is the prominent example for such an NVM. This, however, does not imply that our method is limited to FeRAM; it could be also applied on a non-read-destructive NVM, for instance, by not enabling the read wear-leveling.

The read-destructive property of FeRAM stems from the fact that the reading procedure (i.e., the sensing of a cell value) overwrites the cell [13, 18]. During the read operation, an electric field is applied to the FeRAM cell and the transferred charges are measured, which polarizes the cell. To maintain the original cell state after reading, the old value has to be written to the cell again. This necessary subsequent write access makes FeRAM read-destructive. As the target system, we consider embedded systems in resource-constrained environments, which have to fulfill complex tasks and therefore also run complex software. These kind of systems can be found in automotive controllers or in aerospace applications. Equipping them with NVM is desirable to increase the memory capacity on low costs while maintaining a low energy consumption. In the following, we scope our target system with respect to (1) the cache hierarchy and (2) the general memory architecture.

The class of considered systems usually provides many features that are also available in normal desktop computers. For instance, a complete MMU and virtual memory is often used to isolate the address space of several tasks from each other or to restrict hardware access. However, this does not imply that a full cache hierarchy is possible and useful. The clock frequency of these systems is usually set to some hundreds of megahertz to reduce the power consumption. Memory access latencies become less critical under this condition anyway, and a cache would not improve the situation much but would consume further chip area. Additionally, to guarantee worst-case execution times, scratchpad memory may be preferred over caches. Therefore, the memory wear-out is reduced for the memory regions that are covered by the scratchpad memory, but not for

the other regions. These remaining regions still need a wear-leveling mechanism. In this article, we focus on the worst case that *all* memory regions need wear-leveling. Because of the reasons mentioned previously, our target system consists of an embedded processor with full MMU, virtual memory, and no caches.

To overcome disadvantages of single types of memories (e.g., the lifetime of NVMs or the volatility of SRAMs), several systems implement a hybrid memory architecture [7]. For these systems, more than one memory type is connected to the CPU (e.g., a FeRAM and an SRAM) and mapped to the CPU address space. The operating system and the application then can actively decide which memory content should be placed in which memory, by storing it in the corresponding address region. However, in this work, we only assume one NVM as main memory in the system and develop our solution for all memory segments allocated to this main memory for two reasons. First, if there is a hybrid memory hierarchy with various memories, appropriate maintenance mechanisms for the other memories can be applied separately. We then still provide a wear-leveling mechanism for the NVM part. Second, even if a hybrid memory hierarchy is available, the allocation of memory segments may have to obey several constraints, which makes an arbitrary mapping impossible. Hence, memory contents may still have to be allocated to the NVM, which wears it out rapidly. Our solution provides a mechanism to improve the lifetime of a given mapping of memory segments.

As our proposed methods are software based, they need to run in an operating system like layer to have privileged control over the running application. Even if a full operating system may not be present for small embedded systems, a thin software layer is required to manage the hardware, control startup procedures, and manage the control flow. Our methods can be implemented in such a basic operating system as well. Although we focus on the described target system class throughout this work, our methods are still applicable on other systems with appropriate modifications. For larger systems with caches, for instance, hits and misses would have to be properly distinguished since the first do not wear out the memory but the latter do.

3.1 Implementation Platform

Since we assess our implemented methods regarding their wear-leveling quality in the evaluation, we use a platform for our implementation where we can precisely extract the age (i.e., the total number of accesses per memory cell). We use the full system simulation based framework from our previous work [10]. This framework runs the gem5 simulator in combination with the NVMain plugin for NVM simulations and a special operating system, which allows a sharp separation of application and operating system memory. NVMain outputs a trace file for each simulation that contains precise information about every memory access (i.e., read and write accesses).

Later in this work, we describe our implementation of wear-leveling strategies. We implemented these strategies for the bare-metal operating system, running in the simulation framework as well. Therefore, we can directly evaluate our algorithms in a realistic full system simulation and do not rely on any high-level estimate by analyzing the resulting memory access trace from a simulation with enabled wear-leveling. We further reuse the benchmark applications [10], since the code is directly available with the simulation framework. Nevertheless, our wear-leveling techniques are independent of the CPU architecture, and the concrete implementation and evaluation is done for an ARM-based 64-bit application processor (ARMv8) due to the memory simulator [10]. Note that a concrete implementation on a specific CPU architecture requires several specific implementation details, which are also stated in this article. These details, however, can be reimplemented on other CPU architectures.

4 PROBLEM ANALYSIS

To illustrate the need for wear-leveling and to justify wear-leveling for specific regions, we analyze the memory access behavior of a set of benchmark applications in this section and discuss

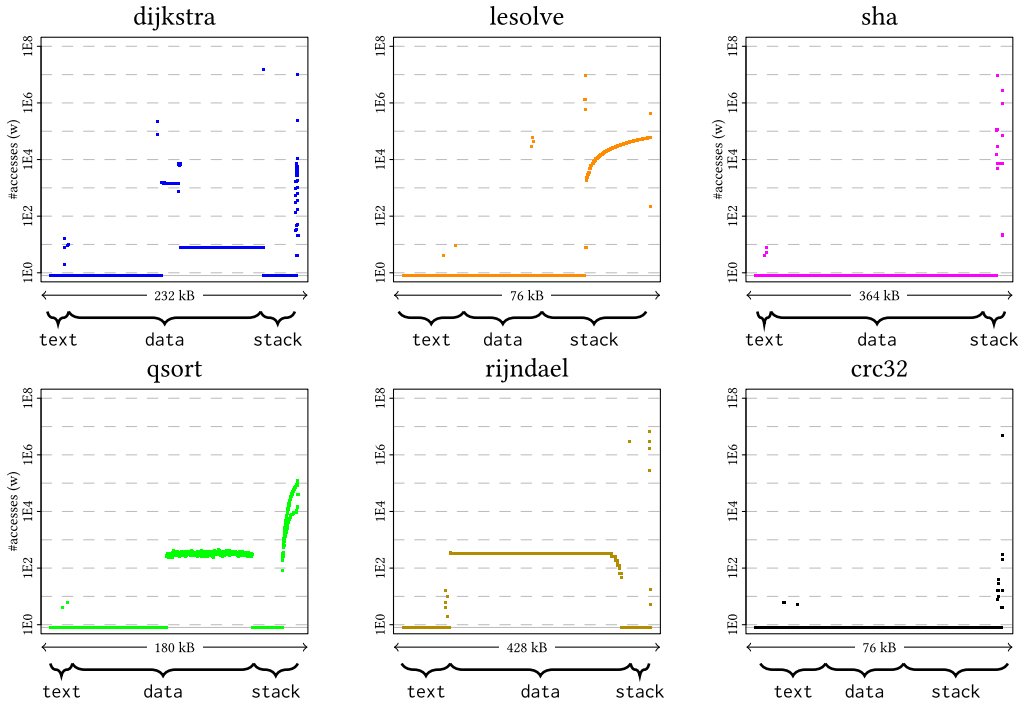

Fig. 1. Benchmark baseline memory traces (write only).

the influence on the memory lifetime. The benchmark applications are presented in detail in the following:

- *dijkstra* is part of the MiBench suite [8] and computes the shortest paths in a graph according to the Dijkstra algorithm. The speciality of this benchmark is that the steps of the algorithm are managed in a queue, which is stored in the data segment.
- *lesolve* is part of the NVM simulation setup [10] and solves a system of linear equations according to the Gaussian elimination algorithm. This benchmark directly modifies its input data.
- *sha* is also part of the MiBench suite [8] and computes the SHA-1 hash of given input data.
- *qsort* is part of the NVM simulation setup [10] and is a recursive implementation of the quicksort algorithm. Therefore, not only is the input data modified, but the stack segment is also used intensively.
- *rijndael* is part of the MiBench security suite [8] and encrypts given input data with the rijndael algorithm. For this benchmark, the input is not read from a file but is read from a region in the data segment itself.
- *crc32* is also part of the MiBench security suite [8] and computes crc checksums on given input data.

Since we target two different scenarios—read-destructive and non-read-destructive NVM systems—we analyze both situations. For non-read-destructive NVM systems, we investigate the total number of write accesses per memory cell, and for read-destructive NVM systems, we investigate the accumulated number of read and write accesses per memory cell. We execute the benchmark applications as described earlier and illustrate the resulting memory access patterns in Figures 1 and 2.

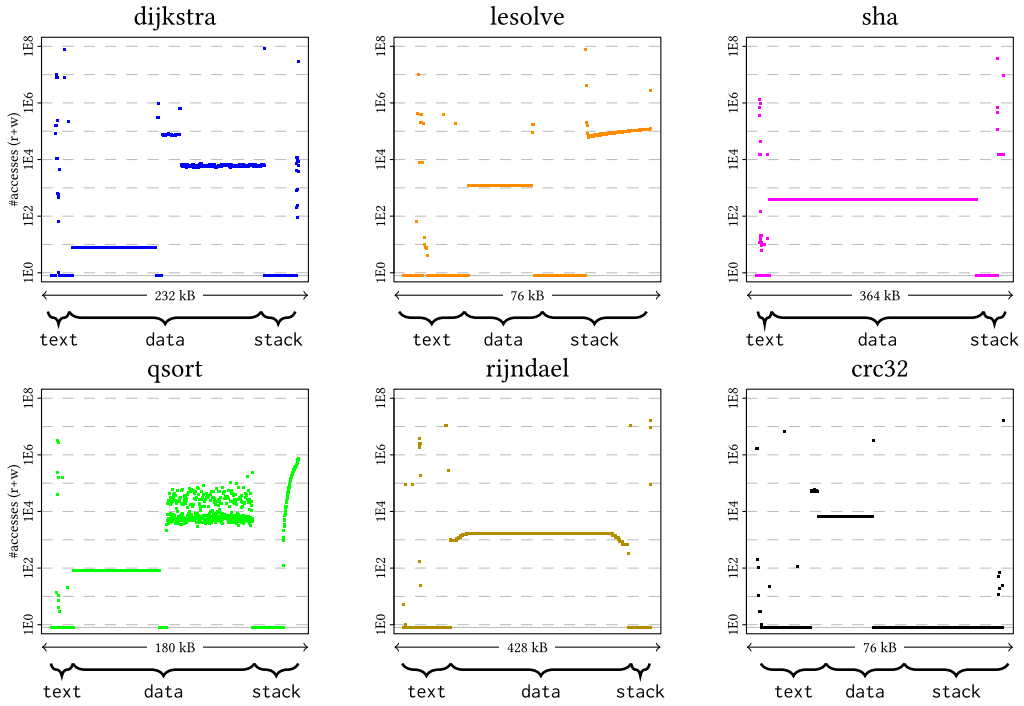

Fig. 2. Benchmark baseline memory traces (write and read).

We observe that memory accesses happen at different rates on memory cells of the different memory regions. Generally, despite large regions with uniform access patterns, dense access hot-spots can be found. These hot-spots have a drastic influence on the memory lifetime, because only a few cells wear out intensively, whereas other cells are not used at all. If these accesses would be better distributed, the lifetime would be increased drastically. For non-read-destructive NVMs (Figure 1), dense write hot-spots are mainly found in the stack, which stems from the way stack memory is used. All other regions face less write hot-spots. For read-destructive NVMs, read hot-spots can be also found in the text regions, because the compiled program code resides in this region and is read during execution.

Overall, we deduce two objectives for our wear-leveling algorithms. First, the regions with different access frequencies have to be detected properly during runtime and have to be relocated to other memory regions, according to the frequency of accesses. We propose a coarse-grained aging-aware wear-leveling algorithm to fulfill this objective. Second, the dense access hot-spots need to be resolved in such a way that the accesses are spread over a larger region of memory cells. This reduces the stress of single cells and averages the wear-out to a larger region. We propose two fine-grained solutions to achieve this: one for the stack segment and another one for the text segment.

5 COARSE-GRAINED WEAR-LEVELING

In this section, we detail the proposed aging-aware coarse-grained wear-leveling. To assess the age of a memory cell, the memory access behavior has to be tracked. If the current access behavior cannot be tracked by the hardware and no memory trace is known for the running application, aging-aware techniques cannot be applied by default. To overcome this issue, we first

propose a software-managed access-distribution approximation technique, which estimates the memory access distribution (i.e., the write and read count to fixed-size memory regions) using only commonly available hardware support (i.e., MMU, performance counters, and interrupts). This access approximation is implemented as a system service in the runtime environment (e.g., the operating system). The access-distribution approximation can be subsequently provided as an input to an aging-aware wear-leveling algorithm.

5.1 Write Access Sampling

As already introduced, the first step toward software-managed coarse-grained wear-leveling is a proper approximation of the memory access distribution. Although capturing this approximation for write and read accesses is mostly similar, we present the capturing of the write approximation in detail first. Subsequently, we describe the additional steps required to also capture the read approximation.

Several steps are required to record an approximation of the write distribution of an application at runtime. First, we equally spaced sample every $C_{sample}^{write}th$ write access of the application, capture its target address, and store it in an appropriate data structure. The number C_{sample}^{write} determines the temporal granularity of the approximation technique, allowing a trade-off between accuracy and introduced overhead. After capturing the write, the spatial granularity of the data structure has to be considered as well. Storing the estimated write count for every byte introduces a big storage overhead and leads to imprecise results, when the temporal granularity is coarse. Instead, bytes can be related to larger memory blocks and the write counts are aggregated for every write access into these blocks. For our implementation, we aggregate the write counts for 4-kB memory blocks, because the wear-leveling algorithm considers this granularity (i.e., the decision is based on memory pages). Using an 8-byte counter for every block, $\frac{1}{512} \cdot memory\text{-}size$ bytes are required to store the approximated write distribution (e.g., 2 MB when 1 GB of main memory is tracked).

The detailed flow of capturing the target of every $C_{sample}^{write}th$ memory write access requires two techniques to be implemented. First, a trap has to be generated after every $C_{sample}^{write}th$ write access, and thus the approximation implementation can take action. Subsequently, the target of a memory write access has to be determined and stored in the data structure. Both implementations are stated in detail subsequently. Although the approach by Gogte et al. [6] allows to directly capture CPU write requests at sampled intervals, their approach relies on a specialized debugging capability. Our method provides an alternative that makes use of more widely available hardware features. Vogl and Eckert [21] propose to use performance counters to specifically analyze instruction execution of an application. We similarly make use of performance counters to analyze memory usage of an application, in contrast, as described in the following.

5.1.1 Temporal Write Distribution Sampling. To generate a trap after every $C_{sample}^{write}th$ write access of the application, we use the CPU internal performance counting mechanism. The `BUS_ACCESS_ST` event in ARMv8 counts the total number of store requests on the memory bus, and thus the number of write accesses of the application is recorded. For Intel CPUs, the same behavior could be achieved by using a performance counter for writebacks of the last-level cache. If no such performance counter is available in some system, any approximation (e.g., the cycle counter or a timer) still can be considered. The performance counting mechanism allows to generate a trap when the performance counter C overflows (i.e., exceeds the value of $C_{max} = 2^{32} - 1$). To establish traps on every $C_{sample}^{write}th$ write access, the performance counter is set to $C_{max} - C_{sample}^{write}$ during the handling of the overflow trap. When choosing the rate C_{sample}^{write}, the introduced overhead for trap handling should be considered.

5.1.2 Write Access Trapping. As the last written memory address cannot be determined during the trap handling of the performance counter overflow, a second technique is implemented to track the target address of the next memory write. During the handling of the overflow trap, the memory access permission for the tracked memory region is set to `READ_ONLY`. Note that the ARMv8 architecture allows hierarchical memory access permissions, allowing to configure memory regions of 1-GB size to `READ_ONLY` by only modifying one page table entry. Due to the `READ_ONLY` permission, the next write access causes a permission violation trap, which is handled as an synchronous interrupt. The violation-causing address is available for the trap handler in a dedicated register, which then is used to increment the corresponding counter in the write distribution approximation.[1] During the handling of the trap, the access permissions are set back to `READ_WRITE`.[2] Note that this mechanism does not strictly require an MMU; it could also be implemented with a very lightweight MPU on a microcontroller. However, if an MMU is present, the write access trapping could be limited to a certain subset of memory pages. If, for instance, some timing critical application relies of fast memory accesses, the write access trapping can be disabled for this application on the cost of bad wear-leveling.

5.2 Read Access Sampling

To record a statistical approximation of read accesses, we follow the same two steps as described before. First, we set up an architectural performance counter that counts read accesses on the memory bus. By setting the performance counter value C manually to its maximum value $C_{max} = 2^{32} - 1$ minus a configurable sampling rate C^{read}_{sample} whenever the counter overflows, an overflow trap is generated every C^{read}_{sample} read accesses. During the handling of the overflow, we set the memory permissions of all observed memory pages to `NO_ACCESS`, which leads to a permission violation trap on read and write accesses. This violation trap is utilized to record the target of the next read access. During the trap handling, the memory permissions are restored such that the execution can continue. In consequence, this mechanism leads to a sampling of the current read address every C^{read}_{sample} read accesses.

In our test system, the read approximation is used alongside the write approximation. Consequently, both methods interfere with each other, since they both use the memory permission system to trap a subsequent memory access. The write approximation only uses the `READ_ONLY` `permission`, and therefore read accesses still proceed and the read approximation is not disturbed. The read approximation in contrast uses the `NO_ACCESS` permission, and thus also a subsequent write access causes a permission violation trap, even if currently no sample for the write approximation should be recorded. This requires tight cooperation between both approximators to ignore these write traps. However, if the read approximator aims to record a read sample but the next memory access is a write access, the write access has to be completed to continue the execution and reach the read request finally.

To complete the write access, the memory permissions have to be relaxed to allow write accesses again. To still trap the next read access, we utilize a debugging mechanism that sets the memory permissions back to `NO_ACCESS` after the write access completed. Therefore, we

[1] The semantics of the performance counter and of the write access trapping mechanism differ slightly. Whereas the performance counter counts every write to the memory, including cache writebacks and other indirect memory accesses, the write access trapping only applies to CPU write operations, which require a fetch of a translation lookaside buffer line from memory. However, this only implies that the distance between two recorded writes is not always C^{write}_{sample}, but sometimes $C^{write}_{sample} + x$, where x is a small integer.

[2] For our runtime system implementation, memory permissions are not used for any protection purposes. If this is the case, the modified permissions might have to be backed up and restored later on.

replace the instruction after the write instruction with a breakpoint instruction.[3] As long as write instructions cannot manipulate the program counter,[4] the subsequent instruction is guaranteed to be executed. The breakpoint handler then replaces the breakpoint with the original instruction, resets the memory permission to `NO_ACCESS`, and continues execution.

5.2.1 Instruction Execution Sampling. When read accesses to main memory are approximated, instruction fetches to the compiled source code should be sampled as well, since they also are memory read accesses. However, using the preceding mechanism would lead to only instruction fetches being captured, since the first thing the CPU does after returning from the trap handler that modified the memory permissions is to fetch the next instruction. Therefore, only accesses to the text segment would be captured in the read approximation.

To overcome this, we do not observe text section pages for the read approximation and therefore do not modify the permissions for these pages. Instead, we take a separate sample of the program counter on every overflow of the performance counter (C^{read}_{sample}), which leads to a separate and independent approximation of the text segment.

5.2.2 Approximation Scaling. As pointed out previously, the read and write approximations are used to estimate the age of memory regions and are fed forward to a coarse-grained wear-leveling algorithm. To maintain the quality of the aging-aware wear-leveling algorithm, it is essential to scale the read approximation according to the write approximation. The read approximation may run with a different sample rate C^{read}_{sample} than the write approximation C^{write}_{sample} for performance reasons. The wear-leveling algorithm, however, only gets the estimated cell age as an input, which is the write approximation for a non-read-destructive NVM and the read approximation plus the write approximation for a read-destructive NVM. Thus, the read and write approximation must have the same weight.

The scaled read approximation c^{read}_{scaled} can be calculated in the following way: the required scaling factor x, which has to be multiplied with the read approximation before it is submitted to the wear-leveling algorithm, is calculated according to Equation (1).

$$x = \frac{C^{read}_{sample}}{C^{write}_{sample}} \tag{1}$$

5.3 Coarse-Grained Wear-Leveling Algorithm

The access-distribution approximation enables arbitrary aging-aware wear-leveling algorithms. The algorithm does not need to be aware if it is running on a read-destructive NVM or not, because read accesses have the same destructive influences as write accesses. Thus, the algorithm can take the age as an input, which is computed from the sum of read and write accesses. We feed the algorithm with an indicator from the access approximation, which estimates the age of each page. Note that the approximation system only operates on virtual memory and does not consider the mapping to physical memory pages. This is maintained by the wear-leveling algorithm itself. The wear-leveling algorithm decides which virtual memory pages are relocated to other physical memory pages and therefore maintains the allover age of the physical memory.

However, the interface between the approximation system and the wear-leveling algorithm has to be well defined. We interleave our wear-leveling algorithm further with the approximation implementation to reduce redundantly stored data. Our wear-leveling algorithm uses a red-black

[3]If the CPU does not provide debug instructions, the same behavior can be achieved by provoking an instruction abort failure due to an invalid instruction.

[4]If the specific CPU allows this feature, this can be still detected by interpreting the current instruction.

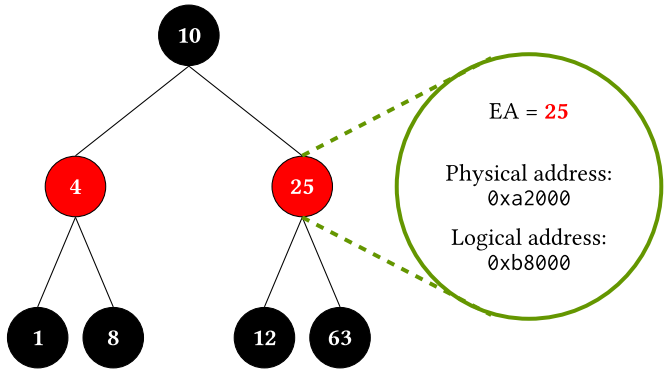

Fig. 3. Organization of the physical memory pages in a red-black tree with their estimated age (EA).

tree to maintain all managed physical memory pages along with their estimated age [10]. As the estimated age is already present inside of the tree nodes, there is no need to store these values in the approximation implementation as well. The tree is illustrated in Section 5.3.1. Each page is stored in the tree with regard to the estimated age, and thus a lookup and extraction of the *youngest* page is efficiently possible. The approximation system maintains a temporary read and write counter per virtual memory page and notifies the wear-leveling algorithm with an age increment action if one of these counters exceeds a certain threshold. In consequence, the wear-leveling algorithm increases the internal age value and relocates the physical memory content to another page.

5.3.1 Management of Memory Pages. Whenever a virtual memory page should be relocated to another physical memory page, the current minimum (i.e., the physical memory page with the lowest assumed age) is extracted from the tree as the target physical page and the estimated ages are adjusted accordingly. The choice of the youngest page as a victim for wear-leveling leads to an incremental wear-leveling, where every page becomes the youngest page after a certain amount of time. Regarding the overhead, the wear-leveling algorithm is only called in this setup when a memory page has to be relocated. Regarding the selection policy of the wear-leveling decisions, the estimated age of all physical pages is balanced equally over time, because every page will be the current minimum page at a certain time. This establishes a stateless incremental wear-leveling, and the memory is assumed to be wear-leveled at any time and is kept wear-leveled. Therefore, the system does not need to store ages across power cycles. The data structures of the access approximation and the wear-leveling algorithm themselves need to be targeted by wear-leveling itself, which requires a special implementation. These technical details, however, are outside the scope of this work.

Eventually, this integration of the wear-leveling algorithm and the approximation system leads to an additional configuration parameter, besides the temporal and spatial granularity of the write-count approximation, i.e. the threshold n_{reloc}, after which number of estimated writes or reads a relocation should be performed. This configuration parameter provides a trade-off between the overhead of page relocation and the frequency, and respectively the resulting quality, of wear-leveling actions without influencing the quality of the access approximation.

5.3.2 Memory Page Relocation. Once the wear-leveling algorithm determines a pair of two virtual memory pages, and respectively their mapped physical memory pages, to swap, two steps are required to perform the relocation. First, the virtual memory mapping in the page table has to be adjusted accordingly such that the physical pages of both virtual memory pages are exchanged. A

translation lookaside buffer (TLB) maintenance operation is required afterward to ensure the exchanged mapping is applied. Note that the ARMv8 virtual memory system allows single entries to be invalidated in the TLB, and thus a total TLB flush is not necessary. After the new page mapping is established, the physical content has to be exchanged to maintain the application's view on the virtual memory. This is achieved by copying one page to a spare buffer, copying the second page to the first page, and copying the buffer content to the second page. The size of the buffer is chosen as 4 kB for two reasons. First, copying a sequential memory content can be done more efficiently in most systems than copying single bytes or words from different regions. Second, the write access pattern to the buffer memory page is completely uniform and thus has no negative influence on the memory lifetime if it is also handled by the wear-leveling system.

6 FINE-GRAINED WEAR-LEVELING

Since the aforementioned algorithm in Section 5 only operates on the granularity of memory pages (4 kB), only the average age of these pages is wear-leveled. In reality, programs use the memory within each memory page very non-uniformly, and thus only a small portion of the page is used intensively. In consequence, leveling the wear on finer granularities has high optimization potential if it manages to wear-level the intensive accesses to single bytes to all the rest of the memory page. Maintaining an aging-aware algorithm as described in the previous section for such fine granularities is not only hard to realize but also causes an immense overhead if estimated ages are stored for single bytes. Therefore, we tackle this problem with non-aging-aware algorithms. These algorithms operate on a small portion of the memory (only a few pages) and wear-level the peak hot-spots within these regions to the entire region. The coarse-grained aging-aware algorithm then still remaps the physical locations of the pages to wear-level them over the entire main memory.

According to various benchmark runs, we identify the stack as the region with the most dense peak hot-spots regarding read and write accesses and the text as the region with the most dense peak hot-spots regarding read accesses. Consequently, we propose two algorithms to wear-level these specific regions internally. Although both algorithms differ in the implementation, there is a common concept—we employ a virtual memory region, called the *shadow region*, which allows us to move memory content within a fixed amount of memory pages in a rotational manner while maintaining full access to all memory contents at all times. We employ this mechanism to move the entire stack and text region within a bounded region of multiple memory pages in small steps (64 bytes in each step). This also moves the dense peak hot-spots in the small steps through the memory and distributes the memory accesses equally. Considering that for our target system the usage of heap memory is not very common, we do not focus on the heap section in this work. If the application uses the heap, however, a similar mechanism as for the stack has to be employed. The rest of this section details the specific implementation for the movement of the stack and text during runtime.

6.1 Shadow Region

An arbitrary piece of memory can be shifted within a larger memory region by copying it bytewise to a new location. This can be also used to move some piece of memory from the bottom to the top of some memory region, which may be a good strategy to spread dense peak hot-spots within the copied memory. However, as long as the memory is in use, the movement is limited because the active memory segment has to be at a consecutive address space and cannot be split. For instance, if 90 bytes are used out of a memory region of 100 bytes, the actively used memory can only be moved by an offset of at most 10 bytes before it would have to be split. To allow a full movement of 100 bytes without splitting the actively used memory, we employ a special virtual

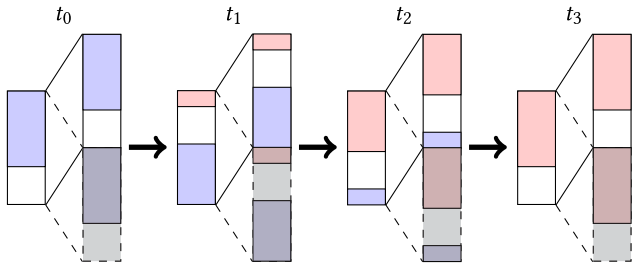

Fig. 4. The physical memory pages (each on the left) and the main and shadow virtual memory map (each on the right) during the movement steps. The colored blocks contain the allocated and used memory; the red color indicates that this block already performed the wraparound.

memory mapping, which we call the *shadow map*. We map the physical pages in the same sequence *twice* into the virtual memory space into subsequent virtual pages.

Figure 4 illustrates the principle of the shadow region. The physical memory pages (each on the left) are mapped twice to consecutive virtual memory pages (each on the right). We call the second virtual memory area the *shadow* because the physical pages are shadowed there from the main virtual memory map. When now the active memory content is moved through the virtual memory, it may cross the boundary between the main and shadow (t_1 and t_2). Still, the entire active memory is fully addressable at consecutive virtual addresses, but the physical content performs a wraparound within the bounded physical memory area. Once the active memory has crossed the boundary entirely (t_4), the wraparound is complete and the physical representation is the same as in t_0. Thus, the system starts to use addresses from the main virtual memory region now instead of addresses from the shadow region. This process is repeated, leading to a rotational movement. As the wraparound is managed in virtual memory, this method does not introduce a large memory capacity overhead. The actual active memory has to be rounded up to multiple memory pages, to ensure the shadow boundary resides exactly between two pages. This method is invasive in the virtual memory system and the memory allocation service of the runtime environment, and thus it has to be ensured that whenever the mapping of either the main or shadow map is modified, the counterpart is modified as well.

6.2 Stack Movement

In combination with the shadow region map, we implement a mechanism to move the actively used stack memory during runtime in arbitrary small steps. We achieve this by copying the stack content to new memory locations. We implement several steps to keep the application's perspective on the stack consistent in this scenario. The stack is relocated from time to time by adding a small offset to the stack pointer (`sp`) and copying the old stack content to the according new location. The logical view of the application always expects free memory bytes before (negative offset) the `sp` and the already created stack content directly after (positive offset) the `sp`. As long as the stack only is relocated within a consecutive memory space, this view can be maintained easily. Due to the employment of the shadow region, a wraparound is achieved while the stack is only moved into one direction. This leads to a rotational relocation of the stack.

6.2.1 Address Consistency. The concept of moving the stack in a circular manner is based on the `sp` relative access of the stack region by C / C++ compiled applications. However, the `sp` relative access is not the only way to access memory contents within the stack memory. Sometimes the application requires to create pointers to variables inside the stack for subsequent function calls

or to store the pointer in a global data structure. Furthermore, pointers to variables on the stack may also be moved out of the stack to some global or heap data structures. During a relocation of the stack, the memory address of the variables on the stack changes, whereas the content of the pointers stays unchanged. This leads to invalid pointers and thus a wrong application behavior. To overcome this problem, we equip the stack relocation system with two pointer adjustment mechanisms, which maintain the correctness of pointer contents over stack relocations.

To provide a mechanism to detect and adjust references to outdated locations within the stack segment, we implement a page-based pointer consistency mechanism. Whenever the stack segment is moved by a small offset d (e.g., 64 bytes), the entire virtual memory location is replaced. Given the stack segment allocates n memory pages, the setup (including shadow) consumes $2n$ virtual memory pages. Instead of relocating from the former base address b to $b + d$, we relocate the stack to the virtual address $b + d + (2n \cdot 4096)$. Due to this, we can invalidate the virtual memory map to the old location of the stack. Whenever the application now holds an outdated address and tries to access it, a trap is raised and handled by the operating system. The trap-causing register is adjusted to the current valid position of the stack segment and the execution can continue. Traps for branches to outdated locations are handled similarly (Section 6.3).

The drawback of this mechanism is that the virtual memory address space is slowly consumed and cannot be reused. However, a simple calculation shows this to be still useful: with a virtual-address size of 48 bits (e.g., for many ARMv8-based CPUs) and 512 MiB being allocated for the system (i.e., cannot be utilized by the consistency mechanism), $2.8 \cdot 10^{11}$ pages are available. When a relocation happens every second and the size of the stack is $n = 8$ memory pages, relocations can continue for 136 years until the system runs out of virtual memory pages. This may exceed the lifetime of most embedded systems by far.

6.3 Text Movement

The second mechanism for fine-grained wear-leveling in this work is a mechanism to move the compiled binary code (i.e., the text segment). This mechanism again employs the shadow region (Section 6.1) to allow a rotational movement of the entire text segment. In contrast to moving the stack (Section 6.2), several different steps have to be performed to maintain the correctness of the program during execution. The basic concept is again to move the text segment in small steps (e.g., 64 bytes) through a subset of memory pages, to distribute the non-uniform read accesses within these pages. To achieve this, we modify the running application to allow to move the binary program code during execution.

6.3.1 Binary Preparation. As a first step toward movable binary program code during execution, we make the entire program code position independent such that it becomes independent of the absolute address of the text segment. This can be achieved by using the *gcc* option `-fPIC`, which generates position-independent code [16]. The resulting compiled binary code performs branches and function calls always relative to the program counter (i.d., to the position of the currently executed instruction). Accesses to global data structures (data and BSS), as well as external function calls, are handled by the **Global Offset Table (GOT)** and the **Procedure Linkage Table (PLT)**. These tables can be accessed with program counter relative addressing. The tables are populated with corresponding absolute addresses from the operating system (i.e., from the dynamic linker) at runtime. The PLT also contains entries for internal functions (not external library functions), since absolute addresses are sometimes used for further address calculation. To avoid any suppression of these entries by the compiler, we compile the application as a shared library and load it into the operating system at runtime. This requires partial linking, where references to external functions and data structures are populated in the GOT and PLT.

6.3.2 Relocation Routine. The actual movement of the text segment in small distances (e.g., 64 bytes) requires the following steps:

(1) Word-wise copy of the binary text
(2) Adjustment of page-based addressing
(3) Address consistency maintenance
(4) GOT/PLT maintenance
(5) PC relocation.

Whereas step (1) is a straightforward copy of single words to new memory locations, the subsequent maintenance steps require some special effort. As mentioned before, we use position-independent code to maintain the independence of the absolute address of the text. For ARMv8, the compiler inserts *adrp* instructions for this purpose (i.e., to address the GOT and PLT), which calculate an address relative to the 4-KiB page of the current program counter. Thus, whenever such an instruction migrates from one to another 4-KiB page, we rewrite the instruction in step (2) and reduce the immediate offset by 1 to maintain the offset calculation to the target. Since the GOT and PLT addresses are always determined by these adrp instruction, we exclude the GOT and PLT from the movement of the text segment. Step (3) employs the same address consistency mechanism as described earlier (Section 6.2.1). Step (4) adjusts self-references to functions and data elements of the application itself to allow the application to still generate correct pointers for these (e.g., function pointers). We finally set the program counter to the according new position and continue execution.

Overall, we provide two specialized mechanisms to move the stack and text in small steps through the main memory. In combination with our shadow region setup, this movement becomes a rotational movement, which spreads dense access hot-spots over a bounded memory region. This the shadow setup operates entirely in the virtual memory space, and the mapped physical pages can be still exchanged by the coarse-grained aging-aware mechanism meanwhile. The implementation only is modified to keep the double mapping of the shadow pages consistent. Thus, allover aging-aware wear-leveling is achieved.

7 EVALUATION

In this section, we evaluate two main scenarios: (1) no-read-destructive NVMs and (2) read-destructive NVMs. For the former, only a subset of the presented concepts is used, and for the latter, all of the presented concepts are employed. For each scenario, we evaluate coarse-grained, aging-aware wear-leveling first. As we already explained, this method cannot achieve optimal wear-leveling, and thus we evaluate it in combination with the fine-grained approaches afterward. First, we detail our evaluation setup and our analysis methodology, then we present the results for our two main scenarios.

7.1 Evaluation Setup

As the technical setup for the evaluation, we use the simulation environment [10], where we also implement our wear-leveling algorithms from Sections 5 and 6. Although the simulation setup executes a full system simulation and therefore our implementation would also run on a real system, using the simulation features the key advantage that we can easily trace memory accesses and analyze them afterward. In this work, we consider byte-addressable non-volatile main memories only (i.e., no block-based memories). Therefore, we only analyze the number of memory accesses per cell and not additional effects, such as block erase in flash-based memories. We record memory access traces for several benchmark applications for a baseline execution without any wear-leveling and for the various combination of employed wear-leveling mechanisms. We always conduct a

full system simulation with a working implementation of the wear-leveling algorithms in the runtime system. We then compare the total number of accesses for every memory byte and compute memory lifetime indicators. For the scenario of non-read-destructive NVMs, we only take write accesses into account, and for read-destructive NVMs, we consider write as well as read accesses. This also implies that the baseline (no wear-leveling) for both of these scenarios is different, and we therefore report improvements in relation to the according baseline. Due to the fact that our implementation is a small bare metal kernel, porting and running applications from known benchmark suites requires manual code integration and the implementation of required system services. Thus, we limit the evaluation to a small set of benchmark applications.

7.2 Analysis Methodology

For each recorded access trace, we aggregate the total amount of filtered accesses to every memory byte. In addition to graphically illustrating the memory access counts over the memory space for our six benchmarks, we consider the analytical gain in memory lifetime. We compute several performance indicators:

- *Achieved endurance*: $AE = \frac{\text{mean\_access\_count}}{\text{max\_access\_count}}$
  Assuming that the memory cannot be used any longer once the first memory cell is worn out,[5] the maximum access count across all cells determines the maximum lifetime. Please note that this circumstance could be omitted by employing additional bad block management. As long as bad blocks are only detected on a more coarse granularity than virtual memory pages, the need for wear-leveling on the granularity of virtual memory pages and smaller granularities still is present. Under perfect conditions, memory accesses could be arbitrarily shuffled to other memory locations to make all cells entirely wear-leveled, which would lead to the mean access count being applied to every cell. Therefore, the quotient of both indicates the percentage of the possible ideal memory lifetime. We do not consider additional spare memory in this evaluation.
- *Endurance improvement*: $EI = \frac{AE_{\text{analyzed}}}{AE_{\text{baseline}}}$
  Given the achieved endurance from the according baseline and another configuration, the quotient of both indicates the improvement in the achieved endurance compared to the baseline.
- *Lifetime improvement*: $LI = \frac{EI}{OV+1}$
  Given the endurance improvement and the overhead OV (percentage of additional memory accesses) of some trace, compared to its baseline, the gained memory lifetime can be calculated by relating both. For instance, if an algorithm improves the endurance by a factor of $EI = 4$ but causes $OV = 100\%$ overhead, meaning that due to wear-leveling the application requires the double amount of memory accesses to complete, the lifetime of the total system is increased by a factor of $LI = 2$.

For all benchmark runs, we calculate the *AE*, *EI*, and *LI* metrics.

7.3 Coarse-Grained Wear-Leveling

Our proposed implementation includes aging-aware coarse-grained wear-leveling, where the age of memory pages is estimated by sampling accesses during runtime. In this section, we only execute the age approximation and the memory page remapping, according to the remapping algorithm (Section 5.3). We record the resulting memory trace and illustrate the total number of accesses per byte graphically.

[5]We note that this assumption may be conservative. Single worn-out cells could be detected and excluded, which would lead to further increased lifetime. However, this requires additional mechanisms.

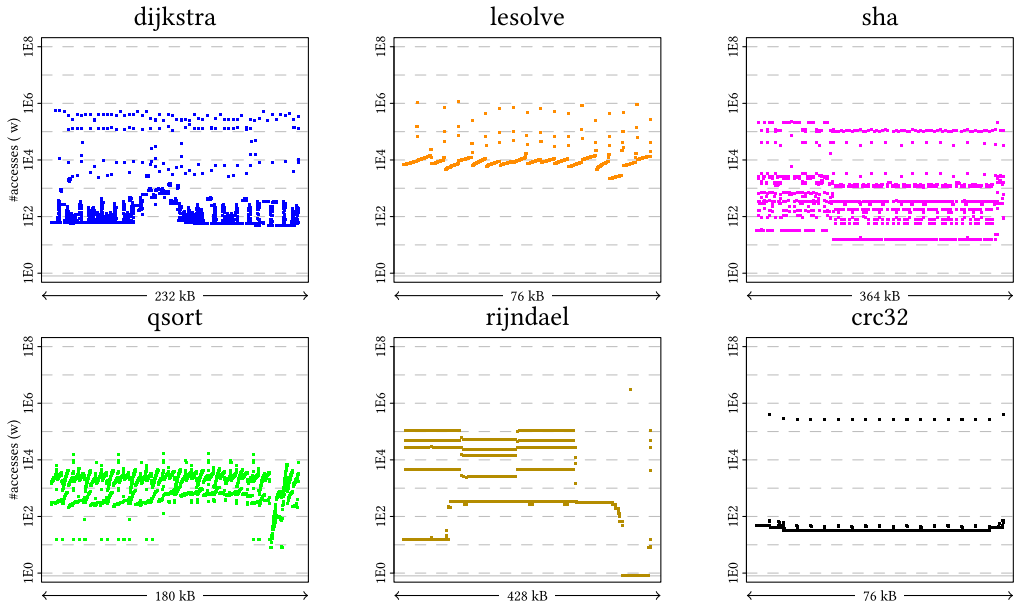

Fig. 5. Coarse-grained wear-leveling (write only).

7.3.1 Non-Read-Destructive NVMs. In case of a non-read-destructive NVM, only write accesses are approximated and the age is only estimated by the number of write accesses per memory page. In Figure 5, we depict the total number of write accesses (y-axis) over the used memory space (x-axis) for our six benchmark applications, when the age approximation and page remapping algorithm are activated. We set the sampling rate of write accesses to $C^{write}_{sample} = 2{,}000$ and the notify threshold for the wear-leveling algorithm to $n_{reloc} = 64$. The results show that the aging-aware algorithm works out because the write accesses are distributed in such a way that all regions of the memory are written with a similar pattern. However, write accesses are still not entirely wear-leveled, which can be deduced from the huge amount of peaks in the figure. It can also be seen that for the benchmarks with larger memory footprints (sha and rijndael), the simulation time was not sufficient to target the entire memory space equally. If the application cannot run for a longer time, the wear-leveling configuration would have to be changed to achieve more frequent wear-leveling to overcome this shortcoming.

7.3.2 Read-Destructive NVMs. When the target system is equipped with a read-destructive NVM, we enable the write and read approximation and estimate the memory page age based on their cumulative amount of read and write accesses, because both are assumed to cause the same wear-out. The wear-leveling algorithm remains unchanged; just the input (i.e., the estimated age) is different. We keep the configuration of the write approximation and the remapping threshold as in Section 7.3.1. We further configure the sampling rate of read accesses to $C^{read}_{sample} = 12{,}000$, because read accesses happen at a much higher ratio than write accesses. Figure 6 illustrates the total amount of cumulative read and write accesses (y-axis) over the memory space (x-axis). An observation similar to that in Figure 5 can be made: the aging-aware wear-leveling works out, even with respect to destructive read accesses. Still, it can be observed that coarse-grained wear-leveling is not sufficient to achieve an allover wear-leveled memory. The applications with larger memory

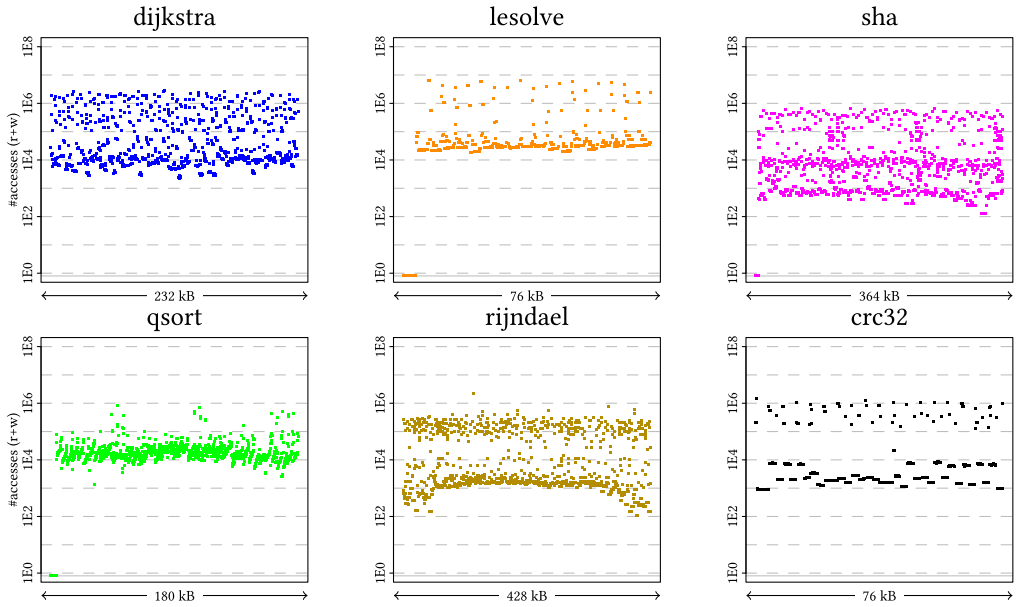

Fig. 6. Coarse-grained wear-leveling (write and read).

footprints result in a better wear-leveling than that presented in Section 7.3.1. Thus, because read and write accesses are encountered, more wear-leveling actions are performed.

7.4 Fine-Grained Wear-Leveling

As the evaluation in Section 7.3 points out, coarse-grained wear-leveling cannot achieve allover wear-leveled memory, since dense access hot-spots within memory pages are not resolved. Consequently, this article proposes additional fine-grained wear-leveling, which is evaluated in this section. We execute the fine-grained stack and text wear-leveling in addition to the coarse-grained wear-leveling to achieve allover aging-aware wear-leveling.

7.4.1 Non-Read-Destructive NVMs. For non-read-destructive NVMs, the fine-grained extension only targets the stack, since the text region is only targeted by read accesses. We keep the same configuration as in Section 7.3.1 and perform a stack movement on every remapping of virtual memory pages (i.e., with the same ratio as the page remapping algorithm). We configure the relocation distance (i.e., the movement of the stack) to 64 bytes. Figure 7 presents the resulting amount of write accesses (y-axis) over the memory space (x-axis). It can be observed that for some benchmarks, a nearly entirely wear-leveled memory is achieved. The shortcoming in the dijkstra benchmark stems from the fact that dijkstra uses the data segment intensively to manage the algorithm steps. Therefore, dense write hot-spots appear in the data segment, which cannot be resolved by our fine-grained stack mechanism.

7.4.2 Read-Destructive NVMs. To perform fine-grained wear-leveling on read-destructive NVMs, dense hot-spots for reads as well as writes need to be tackled. Therefore, in addition to the aging-aware coarse-grained setup from Section 7.3.2, we employ our mechanism for stack and text wear-leveling. We keep the same configuration for the coarse-grained algorithm and execute a stack and a text relocation on every coarse-grained page relocation. In general, both ratios, however, can be separately configured to an arbitrary value. The relocation distance for both stack

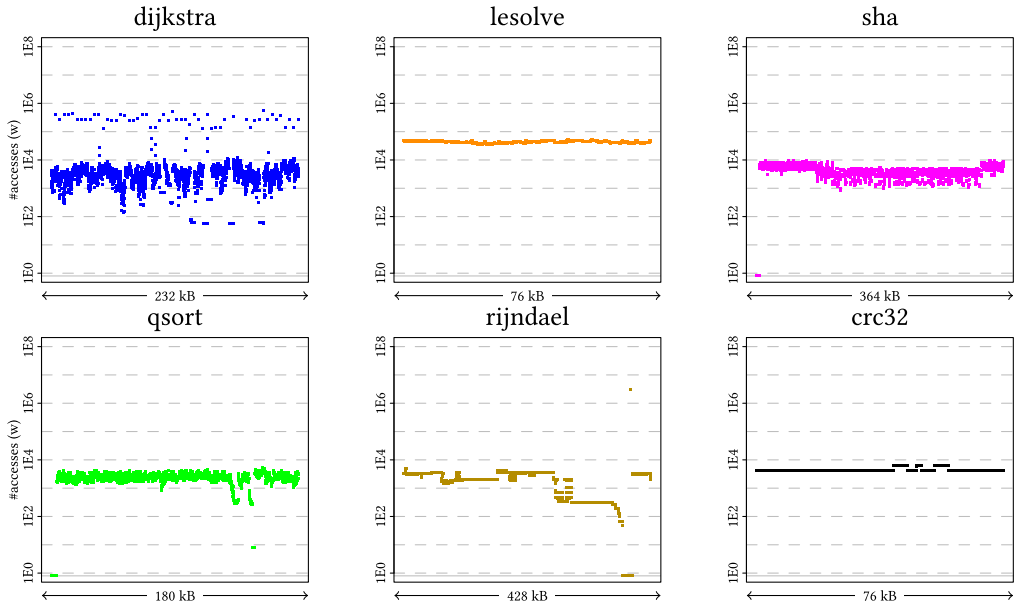

Fig. 7. Fine-grained wear-leveling (write only).

and text relocation is set to 64 bytes. The results in Figure 8 again allow similar observations as for the non-read-destructive case in Section 7.4.1. In general, the memory is allover wear-leveled, considering the destructive influence of reads and writes. For the crc32 and rijndael benchmarks, still larger non-uniformity can be observed. This stems from the fact that the text wear-leveling only moves the relocatable code, but not the GOT and PLT. These two tables, however, are read during the benchmark execution and therefore cause a destructive influence on the underlying memory.

7.5 Analytic Results

Since the figures presented previously only provide an intuition for the achieved quality of our proposed wear-leveling algorithm, we calculate the analytic lifetime indicators (Section 7.2) for all our benchmarks and summarize them in Table 1. Several observations can be made in this table. First, by only considering the last column (*LI*), it can be seen that the total memory lifetime is increased by our algorithm by up to a factor of 955. In other words, a memory lifetime of several days without any maintenance would be extended to many years by only employing our software-based algorithms. Second, read accesses can be slightly worsely wear-leveled than write accesses in some benchmarks, which can be deduced from the lower lifetime improvement. As explained in Section 7.1, another baseline has to be considered for read-destructive NVMs. Thus, the improvement can be significantly lower than for non-read-destructive NVMs. Third, by investigating the first column (*AE*), it can be deduced how optimal the employed algorithms are. If *AE* would be 1, no further improvement would be possible. It can be observed that with coarse-grained wear-leveling only, in most cases only a few percent of the optimal endurance can be achieved. For fine-grained wear-leveling, the algorithms perform significantly better but still allow potential for further improvement. In addition, the achieved endurance differs for the different benchmark applications. The rijndael benchmark achieves by far the worst results, since our algorithms do not handle it properly.

Although the overhead for the various wear-leveling configurations is implicitly included in the *LI* indicator, the overhead as the amount of additional memory accesses due to wear-leveling

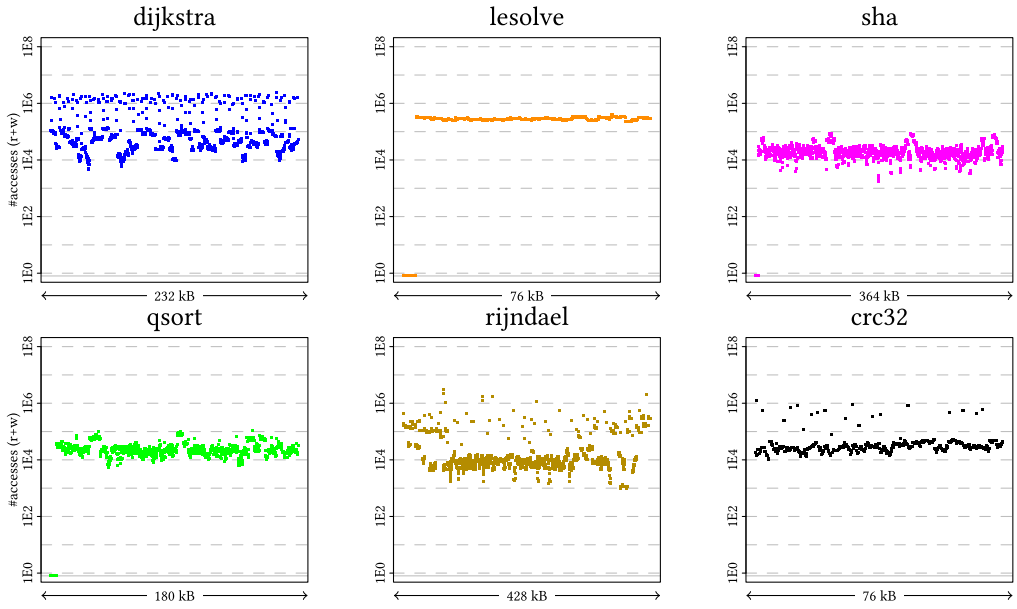

Fig. 8. Fine-grained wear-leveling (write and read).

Table 1. Memory Lifetime Indicators

Application/ Configuration	*AE*	*EI*	*LI*
dijkstra			
-baseline [w]	0.00048		
-baseline [r+w]	0.00110		
-coarse-grained [w]	0.01349	28.1042	27.8912
-coarse-grained [r+w]	0.03283	29.8455	29.5888
-fine-grained [w]	0.01385	28.8542	28.2420
-fine-grained [r+w]	0.04527	41.1545	35.6377
lesolve			
-baseline [w]	0.00189		
-baseline [r+w]	0.00183		
-coarse-grained [w]	0.01712	9.0582	8.9899
-coarse-grained [r+w]	0.02132	11.6503	11.5302
-fine-grained [w]	0.82097	434.3757	189.2855
-fine-grained [r+w]	0.67184	367.1257	194.7539
sha			
-baseline [w]	0.00028		
-baseline [r+w]	0.00033		
-coarse-grained [w]	0.01182	42.2143	41.8878
-coarse-grained [r+w]	0.01796	54.4242	53.8633
-fine-grained [w]	0.42223	1507.9643	955.6642
-fine-grained [r+w]	0.22706	688.0606	418.8788

Application/ Configuration	*AE*	*EI*	*LI*
qsort			
-baseline [w]	0.01609		
-baseline [r+w]	0.00599		
-coarse-grained [w]	0.12174	7.5662	7.5097
-coarse-grained [r+w]	0.02292	3.8264	3.7901
-fine-grained [w]	0.44067	27.3878	23.0112
-fine-grained [r+w]	0.19900	33.2220	27.9929
rijndael			
-baseline [w]	0.00035		
-baseline [r+w]	0.00082		
-coarse-grained [w]	0.00123	3.5143	3.4978
-coarse-grained [r+w]	0.00610	7.4390	7.3514
-fine-grained [w]	0.00088	2.5025	2.3027
-fine-grained [r+w]	0.01431	17.4564	5.1970
crc32			
-baseline [w]	0.00087		
-baseline [r+w]	0.00200		
-coarse-grained [w]	0.01117	12.8396	12.7390
-coarse-grained [r+w]	0.02316	11.5798	11.4619
-fine-grained [w]	0.70932	815.3117	798.0718
-fine-grained [r+w]	0.02764	13.8225	13.0103

Table 2. Memory Access Overhead

Application/ Configuration	*WO*	*RO*	*RWO*
dijkstra			
-coarse-grained [w]	0.764%	0.156%	0.205%
-coarse-grained [r+w]	4.070%	0.591%	0.868%
-fine-grained [w]	2.168%	0.268%	0.419%
-fine-grained [r+w]	87.897%	9.231%	15.480%
lesolve			
-coarse-grained [w]	0.760%	0.210%	0.281%
-coarse-grained [r+w]	3.286%	0.709%	1.041%
-fine-grained [w]	129.482%	19.274%	33.485%
-fine-grained [r+w]	339.729%	51.317%	88.507%
sha			
-coarse-grained [w]	0.779%	0.339%	0.438%
-coarse-grained [r+w]	2.867%	1.155%	1.540%
-fine-grained [w]	57.792%	16.919%	26.111%
-fine-grained [r+w]	140.368%	42.183%	64.262%

Application/ Configuration	*WO*	*RO*	*RWO*
qsort			
-coarse-grained [w]	0.752%	0.174%	0.233%
-coarse-grained [r+w]	3.434%	0.677%	0.958%
-fine-grained [w]	19.020%	2.252%	3.958%
-fine-grained [r+w]	86.406%	11.009%	18.680%
rijndael			
-coarse-grained [w]	0.472%	0.204%	0.253%
-coarse-grained [r+w]	2.836%	0.827%	1.192%
-fine-grained [w]	8.675%	2.023%	3.233%
-fine-grained [r+w]	645.184%	144.883%	235.894%
crc32			
-coarse-grained [w]	0.790%	0.211%	0.287%
-coarse-grained [r+w]	3.302%	0.688%	1.029%
-fine-grained [w]	2.160%	0.417%	0.644%
-fine-grained [r+w]	22.761%	3.746%	6.225%

Table 3. Time Overhead

Application/Configuration	*TO*
dijkstra	
-coarse-grained [w]	6.92%
-coarse-grained [r+w]	20.33%
-fine-grained [w]	14.56%
-fine-grained [r+w]	84.30%
lesolve	
-coarse-grained [w]	6.88%
-coarse-grained [r+w]	17.63%
-fine-grained [w]	67.99%
-fine-grained [r+w]	178.22%
sha	
-coarse-grained [w]	53.95%
-coarse-grained [r+w]	97.08%
-fine-grained [w]	106.79%
-fine-grained [r+w]	245.11%

Application/Configuration	*TO*
qsort	
-coarse-grained [w]	13.67%
-coarse-grained [r+w]	36.51%
-fine-grained [w]	22.93%
-fine-grained [r+w]	87.10%
rijndael	
-coarse-grained [w]	50.33%
-coarse-grained [r+w]	91.09%
-fine-grained [w]	64.35%
-fine-grained [r+w]	588.10%
crc32	
-coarse-grained [w]	6.40%
-coarse-grained [r+w]	15.23%
-fine-grained [w]	8.21%
-fine-grained [r+w]	31.73%

can be investigated itself. When considering performance-sensitive applications, the additional amount of memory accesses makes a major factor for the performance degradation. We calculate the overhead by comparing the total amount of memory accesses from a simulation with wear-leveling to the baseline simulation without wear-leveling. By considering only read accesses, this results in the read overhead (*RO*), for write accesses in the write overhead (*WO*), and for both access types in combination in the read write overhead (*RWO*).

Table 2 contains the resulting overheads for the various wear-leveling scenarios. It can be seen that for coarse-grained wear-leveling only, all overhead types reside at a few percent. When fine-grained wear-leveling is employed, it can be seen that the result of the wear-leveling depends on

the analyzed application. For rijndael, for instance, even allowing huge overheads for the wear-leveling does not lead to significantly increased memory lifetimes. This can be explained by the fact that the wear-leveling does not target these types of memory accesses well. A few intensively accessed memory areas remain not wear-leveled. However, investigating the benchmarks where the wear-leveling can achieve good memory lifetime improvements, the overhead makes up to ≈300%—that is, with wear-leveling, four times as many memory accesses are performed as without wear-leveling. For the interpretation of this result, it should be considered that the overhead can be tuned by the configuration parameter on cost of the wear-leveling result. However, if the application is not performance sensitive, such a big overhead may be still considerable; the memory lifetime is still increased by a factor of ≈200.

The runtime overhead of our wear-leveling algorithms is an important indicator for practical usage. Not only do the additional memory accesses require more time for execution but also the execution of the access approximation and that of the wear-leveling decisions require additional computation time. To analyze this overhead, we compare the total required system cycles for a baseline configuration and configurations with enabled wear-leveling. The relative increase is reported in Table 3. It can be observed that the fine-grained wear-leveling methods in general cause a higher runtime overhead than the coarse-grained methods; the read wear-leveling requires more additional execution time than write wear-leveling. Furthermore, it can be seen that time overheads differ largely for different benchmark applications. For example, crc32 faces an overhead of at most 32%, whereas rijndael faces an increase of the execution time by almost seven times. It should be noted that the time overhead also can be configured by tweaking the frequency of wear-leveling actions. If, however, a performance degradation in terms of execution time of up to almost two times is feasible, most benchmark applications can be wear-leveled using software-managed solutions.

8 CONCLUSION

In this work, we target computer systems that are equipped with NVM as the main memory. We distinguish the cases that this memory is either non-read-destructive or read-destructive. We propose software-managed wear-leveling to improve the lifetime of such systems, since the low cell endurance can cause a severely reduced lifetime. For the former type of system, we take write accesses into account to determine the current age of the memory and to perform according wear-leveling actions, and for the latter case, we take write and read accesses equally into account since both stress the memory equally.

To perform aging-aware wear-leveling (i.e., the current memory age is investigated for each wear-leveling decision) during runtime, we propose a generic runtime approximation of write and read accesses that does not rely on special hardware or debugging capabilities. This approximation is subsequently fed into a wear-leveling algorithm that swaps memory pages according to their estimated age. Since many applications require additional wear-leveling on fine granularities, we further propose two fine-grained wear-leveling mechanisms, where we specifically target the stack and text region. These specific solutions also operate without any special hardware or any special system requirements, and thus they are software managed. The specific solution for the text segment is only invoked for read-destructive NVMs, since the text segment only is targeted by read accesses.

Our evaluation compares the final memory lifetime after applying our algorithms with the memory lifetime of the baseline execution of certain benchmark applications. For non-read-destructive NVMs, we are able to extend the lifetime by up to a factor of 955×, and for read-destructive NVMs, we achieve an improvement of up to a factor of 418×. Although these numbers strongly depend on the memory behavior of the baseline execution of the specific application, we achieve

≈40% of ideal wear-leveling for non-read-destructive NVMs and ≈20% of ideal wear-leveling for read-destructive NVMs. The major shortcomings causing this are memory access patterns that are not explicitly tackled by our methods.

9 OUTLOOK

As our evaluation points out, we achieve a reasonable improvement of the memory lifetime by employing our algorithms accordingly. Still, we cannot achieve the ideal wear-leveling (indicated by the achieved endurance (*AE*)). In other words, our algorithms may be further improved to achieve better wear-leveling in all scenarios. As can be observed for the dijkstra benchmark, the data and BSS sections need specific wear-leveling in some cases as well. In addition, the specific solution for the text segment does not resolve access hot-spots in the GOT and PLT. We intend to improve upon these shortcomings in future work.

REFERENCES

[1] Hoda Aghaei Khouzani, Yuan Xue, Chengmo Yang, and Archana Pandurangi. 2014. Prolonging PCM lifetime through energy-efficient, segment-aware, and wear-resistant page allocation. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design (ISLPED'14)*. ACM, New York, NY, 327–330. https://doi.org/10.1145/2627369.2627667

[2] Chi-Hao Chen, Pi-Cheng Hsiu, Tei-Wei Kuo, Chia-Lin Yang, and Cheng-Yuan Michael Wang. 2012. Age-based PCM wear leveling with nearly zero search cost. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. ACM, New York, NY, 453–458. https://doi.org/10.1145/2228360.2228439

[3] Sangyeun Cho and Hyunjin Lee. 2009. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, New York, NY, 347–357. https://doi.org/10.1145/1669112.1669157

[4] Jianbo Dong, Lei Zhang, Yinhe Han, Ying Wang, and Xiaowei Li. 2011. Wear rate leveling: Lifetime enhancement of PRAM with endurance variation. In *Proceedings of the 48th Design Automation Conference*. ACM, New York, NY, 972–977.

[5] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. 2010. Increasing PCM main memory lifetime. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'10)*. 914–919. http://dl.acm.org/citation.cfm?id=1870926.1871147.

[6] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2019. Software wear management for persistent memories. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 45–63. https://www.usenix.org/conference/fast19/presentation/gogte.

[7] William Goh and Andreas Dannenberg. 2014. *MSP430 FRAM Technology—How To and Best Practices*. Technical Report SLAA628. Texas Instruments. https://www.ti.com/lit/an/slaa628a/slaa628a.pdf?ts=1609843980784&ref_url=https253A252F252Fwww.ti.com252Fproduct252FMSP430FR5989-EP.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the the 2001 IEEE International Workshop on Workload Characteristics (WWC'01)*. IEEE, Los Alamitos, CA, 3–14. https://doi.org/10.1109/WWC.2001.15

[9] Christian Hakert, Kuan-Hsun Chen, Paul R. Genssler, Georg Brüggen, Lars Bauer, Hussam Amrouch, Jian-Jia Chen, and Jörg Henkel. 2020. SoftWear: Software-only in-memory wear-leveling for non-volatile main memory. *CoRR* abs/2004.03244 (2020). https://arxiv.org/pdf/2004.03244.pdf.

[10] Christian Hakert, Kuan-Hsun Chen, Mikail Yayla, Georg von der Brüggen, Sebastian Bloemeke, and Jian-Jia Chen. 2020. Software-based memory analysis environments for in-memory wear-leveling. In *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC'20)*.

[11] Y. Han, J. Dong, K. Weng, Y. Wang, and X. Li. 2016. Enhanced wear-rate leveling for PRAM lifetime improvement considering process variation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 1 (Jan. 2016), 92–102. https://doi.org/10.1109/TVLSI.2015.2395415

[12] Kaixin Huang, Yijie Mei, and Linpeng Huang. 2020. Quail: Using NVM write monitor to enable transparent wear-leveling. *Journal of Systems Architecture* 102 (2020), 101658. https://doi.org/10.1016/j.sysarc.2019.101658

[13] R. E. Jones Jr., P. D. Maniar, R. Moazzami, P. Zurcher, J. Z. Witowski, Y. T. Lii, P. Chu, and S. J. Gillespie. 1995. Ferroelectric non-volatile memories for low-voltage, low-power applications. *Thin Solid Films* 270, 1-2 (1995), 584–588.

[14] Qingan Li, Yanxiang He, Yong Chen, Chun Jason Xue, Nan Jiang, and Chao Xu. 2014. A wear-leveling-aware dynamic stack for PCM memory in embedded systems. In *Proceedings of the 2014 Design, Automation, and Test in Europe Conference and Exhibition (DATE'14)*. IEEE, Los Alamitos, CA, 1–4.

[15] Wei Li, Ziqi Shuai, Chun Jason Xue, Mengting Yuan, and Qingan Li. 2019. A wear leveling aware memory allocator for both stack and heap management in PCM-based main memory systems. In *Proceedings of the 2019 Design, Automation, and Test in Europe Conference and Exhibition (DATE'19)*.

[16] ARM Limited. ARM Compiler User Guide (Version 6.12). Retreived November 20, 2021 from https://static.docs.arm.com/100748/0612/compiler_user_guide_100748_0612_0 0_en.pdf?_ga=2.51313322.225031596.1586250715-64667359.1569664146.

[17] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha. 2013. Curling-PCM: Application-specific wear leveling for phase change memory based embedded systems. In *Proceedings of the 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC'13)*. 279–284. https://doi.org/10.1109/ASPDAC.2013.6509609

[18] E. M. Philofsky. 1996. FRAM—The ultimate memory. In *Proceedings of the Nonvolatile Memory Technology Conference*. 99–104. https://doi.org/10.1109/NVMT.1996.534679

[19] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. 2009. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 14–23. https://doi.org/10.1145/1669112.1669117

[20] Songping Yu, Nong Xiao, Mingzhu Deng, Yuxuan Xing, Fang Liu, Zhiping Cai, and Wei Chen. 2015. WAlloc: An efficient wear-aware allocator for non-volatile main memory. In *Proceedings of the 2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC'15)*. 1–8. https://doi.org/10.1109/PCCC.2015.7410326

[21] Sebastian Vogl and Claudia Eckert. 2012. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of the 2012 European Workshop on System Security (EuroSec'12)*, Vol. 12.

[22] W. Zhang and T. Li. 2009. Characterizing and mitigating the impact of process variations on phase change based memory systems. In *Proceedings of the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 2–13.

[23] M. Zhao, L. Shi, C. Yang, and C. J. Xue. 2014. Leveling to the last mile: Near-zero-cost bit level wear leveling for PCM-based main memory. In *Proceedings of the 2014 IEEE 32nd International Conference on Computer Design (ICCD'14)*. 16–21. https://doi.org/10.1109/ICCD.2014.6974656

[24] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 14–23. https://doi.org/10.1145/1555754.1555759

Received February 2021; revised June 2021; accepted August 2021