technische universität
dortmund

OCTO$^+$: Optimized Checkpointing of B+ Trees for Non-Volatile Main Memory Wear-Leveling

Christian Hakert, Roland Kühn, Kuan-Hsun Chen, Jian-Jia Chen, Jens Teubner
Technische Universität Dortmund
Design Automation for Embedded Systems Group / Databases and Information Systems Group

Citation: To be published at NVMSA 2021

BibTeX:

```
@inproceedings{Hakert2021octo,
    author = {Hakert, Christian and K\"uhn, Roland and Chen, Kuan-Hsun and Chen, Jian-Jia and
    Teubner, Jens},
    title = {OCTO+: Optimized Checkpointing of B+Trees for Non-Volatile Main Memory Wear-Leveling},
    booktitle = {The 10th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)},
    year = {2021},
    publisher = {IEEE}
}
```


CS 12 computer science 12

OCTO$^+$: Optimized Checkpointing of B$^+$ Trees for Non-Volatile Main Memory Wear-Leveling

Christian Hakert, Roland Kühn, Kuan-Hsun Chen, Jian-Jia Chen, Jens Teubner
Technische Universität Dortmund
Design Automation for Embedded Systems Group / Databases and Information Systems Group

***Abstract*—Steady deployment of byte addressable non-volatile memories (NVMs) as main memory in server class computers yields challenges on software. In order to overcome shortcomings, as for instance low cell endurance and high access latencies, working data can be kept in DRAM and continuously be checkpointed to the NVM. Although this reduces the impact of the NVM on usual execution, it shifts the endurance and latency issue to the checkpointing. Alongside widely studied generic wear-leveling solutions, we propose an application cooperative wear-leveling scheme for B$^+$ trees, that realizes an interplay of the application and the wear-leveling. We collect memory usage statistics during tree operations and dynamically choose a memory mapping between the DRAM footprint and the NVM checkpoint of the B$^+$ tree. In an experimental evaluation, we achieve $3\times$ improvement in terms of memory lifetime.**

***Index Terms*—non-volatile memory, b-tree, wear-leveling, checkpointing, software based**

I. Introduction

Continuous improvement in the field of non-volatile memory (NVM) as main memory leads to market ready solutions throughout the last years. Embedded Systems with on-chip byte addressable NVM are widely available (e.g. the *MSP430FR* family from Texas Instruments), but also servers with large amounts of NVM in the form of additional memory modules, as for instance the Intel Optane PMem, are available. The various challenges, NVM main memory imposes on the system and the software, have been widely studied during the last years on a generic level to provide application transparent maintenance solutions. A possible interplay of specific applications with the maintenance application itself and therefore an application-specific scheme, however, is rarely considered due to heterogeneity of applications.

In this paper, we explore B$^+$ trees, executed on a hybrid memory system, where the B$^+$ tree is persisted in NVM and executed and modified in DRAM. As B$^+$ trees play a central role of database systems, improvement of memory lifetime, latency, and energy efficiency can be eventually beneficial for various applications. To this end, in contrast to generic wear-leveling schemes, we investigate application cooperative wear-leveling as one instance of maintaining NVM lifetime with low overheads, where we explicitly modify the application and realize an interplay of the application and the wear-leveling scheme to achieve aging-aware software-based wear-leveling.

For such a hybrid memory system, the updates of B$^+$ trees are temporarily operated in DRAM but have to be finally applied to the NVM at *checkpoints*. This so-called *checkpointing* mechanism, which may cause unnecessary writes and bit flips if it is not done carefully. One simple solution is to perform checkpointing periodically, in which a copy of the B$^+$ tree is created and written to the NVM to ensure persistency. This simple strategy reduces the total amount of write accesses to the NVM, but limited endurance on a NVM remains an issue. As an example, consider a system which performs checkpointing every minute to a phase change memory (PCM) NVM, whose endurance is 10^6 write cycles [5], a memory cell written at every checkpoint wears out within 2 years.

In this paper, we present an application cooperative wear-leveling approach for the checkpointing of generic B$^+$ trees, which introduces only minimal overhead by re-evaluating and modifying a mapping table, which maps B$^+$ tree nodes to blocks within the checkpoint. Our target is not only to increase the memory lifetime but also to groom the memory space for improving the application of further generic wear-leveling solutions. We modify the B$^+$ tree implementation to track modifications to single nodes, which are applied between two checkpoints. We further use this information to estimate the individual aging of NVM locations afterwards. Based on this knowledge, we run a remapping algorithm, modifying the mapping in order to optimize the wear-out. The remapping algorithm runs in linear time complexity with the memory size of the B$^+$ tree.

Our contributions:

- We provide a modified implementation of a generic B$^+$ tree, which tracks modification information of tree nodes during execution.
- We further provide an application cooperative wear-leveling algorithm, which utilizes the collected information to improve the lifetime of the NVM device by modifying a mapping of regularly created checkpoints.
- We conduct a precise evaluation of memory wear-out on cell granularity and estimate the potential for further generic wear-leveling.

II. Related Work

The use of NVM for indexes in databases is extensively studied in the last years. However, most work has been dedicated to exploit features such as byte addressability. The problem of higher latency compared to DRAM, especially during writes, is thereby addressed by many researchers in this effort, leading to new index structures that mainly try to avoid unnecessary writes to NVM. Many of these approaches

978-1-6654-2375-5/21/$31.00 ©2021 IEEE

attempt to reduce data movement within leaf nodes. Some of them allow unsorted leaf nodes and maintain additional helper structures to improve the performance of search operations [7], [20], [25]. Other approaches enhance search operations by dividing leaf nodes in cacheline-sized chunks that are sorted internally, even if the node itself is not sorted [24]. Still other approaches allow unsorted inserts into leaf nodes, whilst the nodes are sorted occasionally [2].

While writing to NVM is identified as a critical problem in almost all of the approaches mentioned above, the focus is primarily on reduced write performance rather than wear-leveling. To our knowledge, the aspect of wear-leveling, and in particular checkpoint-based wear-leveling, plays only a minor role for indexes so far. Chi et al. [8] propose a cost model for B^+ trees in their work that estimates wear out of nodes, but they do not focus on wear-leveling.

Contrarily, generic in memory wear-leveling for NVMs is widely studied in the literature. These approaches can be classified into aging-aware approaches and non-aging aware approaches. The former gather the information about the memory wear-out either directly from the memory hardware [1], [6], [10], [28], use software based approximations [11], [12], [14], [17] or hook into the memory allocation process [1], [19], [26]. Also non aging-aware approaches exist, which apply blind or random memory maintenance [11], [23]. All of these aging-aware approaches, however, do not aim to modify the application itself. Some approaches try to take application characteristics into account [13], [15], but these approaches are not specific for a certain application.

III. System Model

As mentioned earlier, it may be desirable to equip a system with large amounts of NVM due to its low cost and power consumption. However, reduced cell endurance and higher read and write latencies make solely use of NVM as main memory a suboptimal choice. The read latency of phase change memory (PCM), for instance, is reported to be 2x - 6x higher as for DRAM, the write latency even 2x - 15x [4]. The cell endurance is reported to be as low as 10^6 or 10^7 write cycles [5], [18], which is $10^8 - 10^9$ less than the expected endurance of DRAM. Therefore, we consider a system with both, DRAM and NVM as main memories in this work. Both memories are byte addressable and mapped to the physical address space of the system, such that software freely can decide which content to place in which memory. We further assume unaligned memory accesses to be supported by the memory hardware, such that unaligned copies from the DRAM to the NVM do not cause a high overhead compared to aligned copies. In a set of micro benchmarks on different architectures, we observe a performance impact of not more than 30% when copying sequential data unaligned.

Iterative write schemes are widely considered for PCM [18], [22], [27], as they reduce the latency and increase the lifetime. We therefore investigate the memory write trace for the NVM and count bit flips on every single bit to assess the memory wear-out. In this paper, we consider the cumulative count of bit flips per memory cell to assess the wear out of these cells. Some recent results, e.g., [18], demonstrate that the different pulses for set and reset in PCM may cause different stresses on the memory cells. However, we assume that over the lifetime every single cell faces approximately similar number of bit flips from 0 to 1 and from 1 to 0. That is, the amortized impact can also be modeled approximately by the cumulative count of bit flips.

IV. B^+ Tree Checkpointing

In order to exploit the low latency and high endurance of DRAM, we store a background copy of the B^+ tree entirely in NVM, but cache a working copy in DRAM. Therefore, we do not consider NVM specific modifications of B^+ trees, such as [2], [7], [20], [24], [25] and stick to a generic implementation. In this paper, we focus on the B^+ tree only, managing the data records itself properly within the NVM is out of scope of this work. We assume furthermore that the tree is held completely in DRAM, as typically done by in-memory databases. With a certain ratio, the memory content of the B^+ tree is copied to the NVM, which ensures persistency. We intentionally store the entire tree in the NVM and do not aim to rebuild it during restoring to reduce the time requirement for restoring and to allow further optimizations, such as copy on write. Tree modifications between two *checkpoints* result in a single write of the modified memory to the NVM. The writing of checkpoints itself performs write accesses to the NVM and causes wear-out, which leads to the need of wear-leveling the checkpoints.

Problem: As a mechanism to wear-level checkpoints, the memory layout of the checkpoint can be modified by applying a remapping table, which assigns every node of the B^+ tree to a new position in the NVM. This remapping table can be stored in a metadata block of the checkpoint. The metadata block can be handled as any other block for wear-leveling, only the offset of the metadata blocks has to be kept centrally by the system. In general, modifications of the mapping introduce a high additional cost since potentially unchanged B^+ tree nodes can be mapped to a new NVM location. This not only *cause additional write accesses* and therefore a higher time demand of the checkpoint, but also *a lot of bit flips in the NVM* and thus additional wear-out. Thus, a wear-leveling algorithm needs to keep the mapping untouched whenever possible and carefully modify the mapping if required.

Objective: To achieve this in an efficient way, knowledge about the modifications within tree nodes is crucially required before every checkpoint. Modifications of the DRAM copy of the B^+ tree can be either gathered with the help of special hardware support [1], [6], [10], [27], [28], or by software based tracking mechanisms, on the cost of overheads [11], [12], [14], [17]. In this work, we propose an alternative lightweight scheme, in which we collect the information about modified memory contents specifically for B^+ trees during the execution of the tree operations (insert, update, delete, lookup) itself. This mechanism is not bound by any limitation on temporal and spatial granularity of a generic mechanism and can be

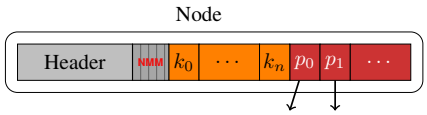


Fig. 1: Layout of a node, the different colors reflect different memory regions

realized with minor code modifications instead of employing an entire additional hardware or software layer.

We consider to manage the DRAM and NVM in blocks, which are equally sized as the B$^+$ tree nodes. We call a DRAM block *modified* if modifications were applied since the last checkpoint, i.e. the checkpoint potentially causes bit flips in the NVM. We call a NVM block *old* or *young*, representing the remaining lifetime. We further denote DRAM and NVM blocks as *uniform modified* / *uniform aged* if the stress or the age is similar for all bits of the block.

V. OCTO$^+$ Algorithm

In this section, we present OCTO$^+$, an algorithm carefully modifying the checkpoint mapping to achieve wear-leveling. After collecting modification information directly within the B$^+$ tree, the algorithm features two wear-leveling targets. First, the algorithm performs *intra block* wear-leveling, which aims to detect not uniform aged NVM blocks and map B$^+$ tree nodes in such a way to them during future checkpoints, so that they become uniformly aged again. Second, the algorithm performs *inter block* wear-leveling, which aims to balance the absolute age of all NVM blocks.

A. Write Information Collection

Since read operations do not modify any data that is stored in the B$^+$ tree, the memory usage statistics are just collected during insert, update or delete operations. A node of our B$^+$ tree (see Figure 1) can be divided into three contiguous memory regions. The first region contains the header of the node. The header stores all information like fill level and the type of a node (leaf node or inner node) that are important for tree-internal operations like splits, merges or redistribution. The second region contains an array of keys, while pointers to child nodes or values are stored in an array in the third region. To keep the collection of the modified parts of a node as simple as possible, we extended the header with a node modification mask (NMM), which is a fixed width bitmask. The insertion of the NMM slightly enlarges the header. However, given a fixed size for each node this may only result in a slightly lower capacity of keys and values/pointers per node. In the following we assume that this bitmask contains 8 bits. The mask indicates which parts of the node were modified after the last checkpoint and therefore must be considered for the next checkpoint. The modifications to the different procedures for inserts or updates are also moderate. Every time a key/pointer or key/value pair is inserted, we set the corresponding bit that is mapped to the position of the key and the value/pointer array in the node modification map. Even if this approach is coarse grained, the kind of meta information that is stored in the node itself can be easily extended to allow a more fine grained partition of the node or to adapt our approach to other algorithms and data structures. It should be noted that although the node modification mask is stored in the B$^+$ tree nodes itself, it does not need to be written out to a checkpoint and therefore modifications to the node modification mask itself do not need to be considered in the context of wear-leveling.

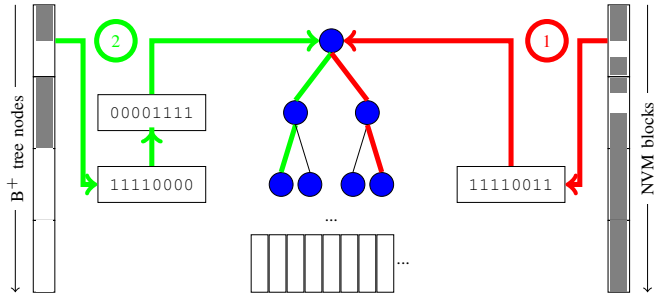


Fig. 2: OCTO$^+$ tree based algorithm[1]

B. Remapping Decisions

The mapping of B$^+$ tree nodes to NVM blocks is re-evaluated on every checkpoint and potentially updated. The decision for modifications of the mapping ① identifies not uniform aged NVM blocks and remove them from the current mapping, ② maps unmapped tree nodes to NVM blocks and ③ updates the internal age counters of modified NVM blocks.

We maintain an internal aging representation of the NVM blocks in the form of eight 8-bit counters, where each counter indicates the age of one eighth of the NVM block. These counters are stored in a single 8-byte CPU word and modified with bitwise logic operations. The counters are updated with the modification information of the tree nodes in step 3. Since our employed wear-leveling scheme is incremental and aims to achieve a wear-leveled memory at any time, the internal aging representation does not need to be checkpointed itself. If this information is lost, the memory can be assumed to be wear-leveled and all counter values can be reset to 0.

1) Intra Block Wear-Leveling: In order to identify not uniform aged blocks, the aging representation of all currently used NVM blocks is checked. If any of the 8 counters within each block exceeds the average of the block by a certain threshold, this block is released from the mapping. This procedure results in a set of unmapped B$^+$ tree nodes and unmapped NVM blocks. In the next step, unmapped tree nodes are mapped to the previously released NVM blocks. We construct an 8-bit bitmap for every NVM block, where a 1 bit indicates that the corresponding counter is greater than the mean of all counters for this block and a 0 bit indicates a counter lower than the mean, respectively. According to this bitmap, the spare NVM blocks are stored in 256 lists, which are indexed by

[1]The B$^+$ tree nodes are indicated on the left, gray areas are modified sections, white areas are unmodified sections. The NVM blocks are indicated on the right, gray areas are aged section, white areas are young sections, respectively. For the tree node, the 8-bit bitmap is inverted. The central mapping tree datastructure is used to store unmapped NVM blocks in step 1 and assign tree blocks in step 2.

a binary tree of depth 8, called *mapping tree*, where the left child indicates a 0 bit and the right child indicates a 1 bit. In consequence, insertion of blocks into this tree requires a constant overhead of 8 decisions. Additionaly, each node of the mapping tree maintains a counter, how many NVM blocks are currently stored in lists underneath that node. Afterwards, the unmapped tree nodes are mapped to NVM blocks by traversing the tree with the bitwise inverse node modification mask. In total, this procedure maps tree nodes to NVM blocks in such a way, that modified regions of the tree node are placed on young parts of the NVM. This two step procedure is illustrated in Figure 2. When mapping tree nodes to NVM blocks, the block counter within each node of the mapping tree makes sure that searching for a spare NVM block does not end at an empty list, even though this may violate the optimality of the mapping.

After the mapping is updated according to the aforementioned procedure, the aging representation of the NVM blocks is updated by incrementing the 8-bit counters with the corresponding bit of the node modification mask. If any counter reaches the maximum value, all counters for all NVM blocks are divided by 2. This keeps the relative aging information in the aging representation. To avoid thrashing within the mapping, remapped NVM blocks are excluded from the first step for a certain amount of subsequent checkpoints. We set this amount to the aforementioned threshold, which is used to decide if a block should be released from the mapping.

2) Inter Block Wear-Leveling: The mapping decision as detailed in Section V-B only aims to level uneven aging patterns within tree node sized NVM blocks by carefully updating the mapping during the checkpointing. Even though this method improves the NVM lifetime already, it may still happen that some NVM blocks face a higher total amount of modifications than other blocks. To also tackle this situation, we equip our remapping decision with an additional aging aware scheme. During every checkpoint, the maximum of the 8 counters for each NVM block is determined. If the difference of these values for the youngest and the oldest block exceeds a configurable threshold, the mapped B^+ tree node of both of these blocks is exchanged. Due to the fact that we divide all counters for all blocks on an overflow, also the relation of the absolute aging of NVM blocks remains. Determining the youngest and oldest blocks does not induce much additional overhead since during a checkpoint anyway all blocks have to be traversed once for the intra block wear-leveling.

C. Static Optimization

The OCTO$^+$ algorithm, as described before, aims to tackle intra and inter block wear-leveling. This, however, is done by approximating the memory age and modification with the help of a coarse 8-bit bitmask. A non uniform memory usage below the granularity of node size divided by 8 therefore cannot be handled. A special case of this fine grained non unformity is uneven usage within single words. To overcome this, we shift every B^+ tree node during checkpointing by a fixed offset between 0 and 7 bytes. This requires unaligned memory access during the checkpoint, as mentioned in Section III. This method requires no further metainformation, since we determine the shift offset by calculating the node id modulo 8. As the shift offset does not change over time for every node, this also does not introduce any extra memory write overhead.

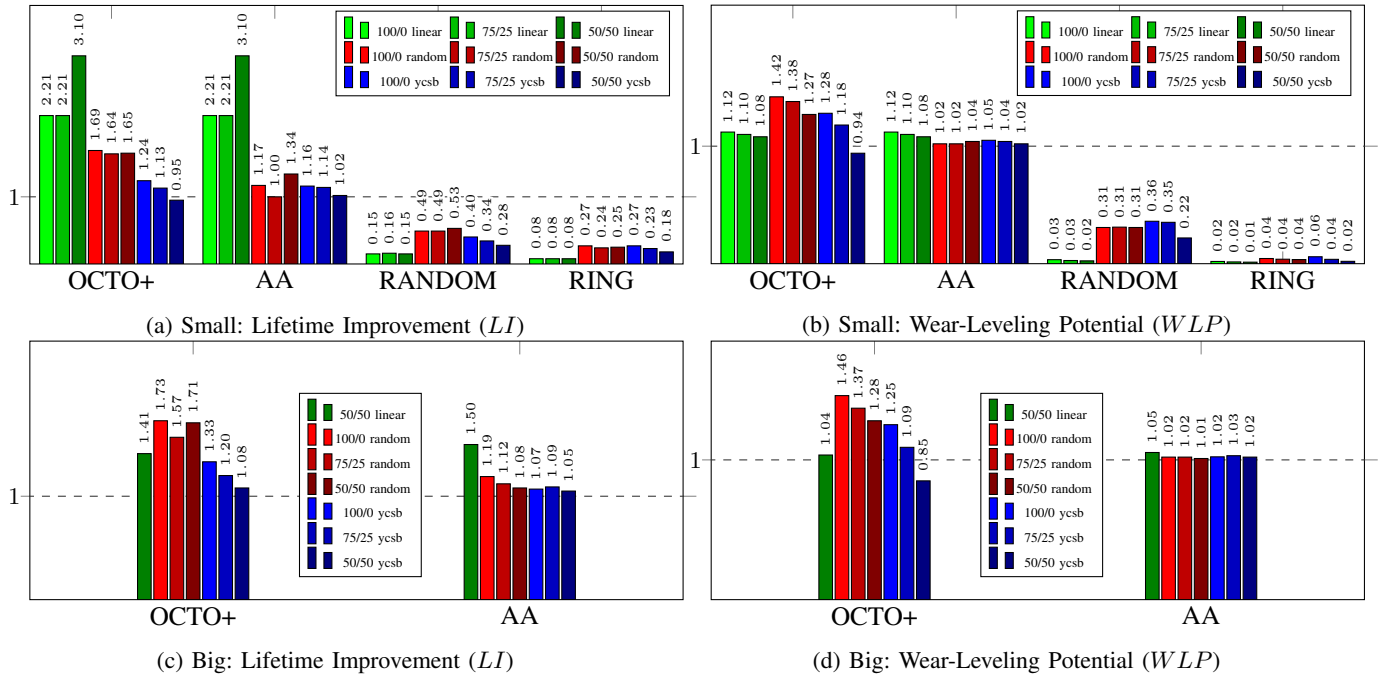
VI. Evaluation

In order to evaluate the improvement in terms of lifetime of the NVM of the algorithm presented in this paper, we implement it in a full system simulation setup [16]. This allows us to run the B^+ tree, the mapping algorithm and also the checkpointing on a simulated ARMv8 processor [3]. The additional employment of the NVMain2.0 plugin [21] further allows to trace each and every memory access to a trace file and analyze it with regards to the amount of flips per bit afterwards.

A. Evaluation Setup

For our evaluation we choose a node size and therefore a block size of 1024 Bytes for our tree, since it offers a good compromise between the fanout of a B^+ node and the granularity of the write information collection. We then evaluate the tree with three different data sets with a key and value size of 8 byte each. The first data set contains values in a monotonic order. This data set simulates the creation of a B^+ tree on presorted data, which results in a tree that contains many half filled nodes. Due to the monotonic order of the keys, older leaf nodes will not be modified anymore. The second data set consists of random values, which leads to more modifications also of older nodes. Furthermore, we choose keys from YCSB [9] for the third data set.

Based on these data sets, we investigate two different tree sizes and 3 different insert and update distributions each. We consider a *small* tree, at which we perform 20000 operations and a *big* tree, at which we perform 50000 operations. To simulate insert or update heavy workloads, we further split the operations into either 100% inserts and 0% updates, 75% inserts and 25% updates or 50% inserts and 50% updates. In every configuration, we perform a checkpoint after 50 tree operations for the small tree and after 100 tree operations for the big tree, which leads to a total amount of 400 checkpoints for the small trees and of 500 checkpoints for the big trees. As a baseline, we consider a static mapping for the checkpoint, which remains unchanged. In comparison, we run the checkpointing with our intra and inter block remapping algorithm, which we call OCTO$^+$ in the following. We further compare to only the inter block wear-leveling without intra block wear-leveling, which is denoted as aging aware (AA) in the following. In addition, we implement a random modification of the mapping table, called RANDOM, at every checkpoint and another strategy, which inheres a ring based remapping scheme [23]. This strategy, called RING, applies a constantly increasing shift offset to every NVM block during the checkpoint and wraps around at the end of a block. Due to preliminary experiments, we set the threshold for the



(a) Small: Lifetime Improvement (LI)

(b) Small: Wear-Leveling Potential (WLP)

(c) Big: Lifetime Improvement (LI)

(d) Big: Wear-Leveling Potential (WLP)

Fig. 3: Wear-Leveling Improvement

unmapping of not uniform aged blocks in intra block wear-leveling to 15, the threshold for inter block wear-leveling to 5, respectively. These experimentally determined parameters achieve adequate wear-leveling results while not stressing the NVM with many additional bit flips.

B. Evaluation Metrics

Assessing the improvement in terms of memory lifetime is based on two separate considerations in this work. We assume that the NVM becomes unusable once the first cell wears out. This could be resolved with spare cells and detection of dead cells, however, such static improvements can be applied independently. Under this assumption, we assess the maximum amount of bit flips over the entire allocated NVM space for the B$^+$ tree. We compare this *peak age* for the different configurations and compute an improvement factor LI, which compares to the corresponding baseline (unmodified mapping).

Although the above metric reports the theoretic lifetime improvement if the B$^+$ tree and checkpointing algorithm runs in isolation on the limited memory, global wear-leveling across many applications is not considered.

$$WLP(g) = mean\left(\max_{[0,g)}(age(x)), ..., \max_{[m \cdot g,n)}(age(x))\right) \quad (1)$$

We analyze the potential, our algorithm delivers for such a global scheme by considering the mean value of the ages of NVM regions on a given granularity. Provided with the aging of each region, a perfect global wear-leveling could arrange regions in such a way, that exactly the mean age is applied to all regions. Therefore, with a lower mean age a higher lifetime can be achieved by a global wear-leveling scheme. We compute this wear-leveling potential WLP by Equation (1). m denotes the number of NVM regions, g the granularity, respectively. For every configuration, we compute this number with the granularity of memory pages ($b = 4096 \cdot 8$), since generic wear-leveling techniques potentially utilize the MMU to remap 4kB memory pages. The ratio of the WLP metric of a configuration with the corresponding baseline then indicates the improvement in wear-leveling potential.

C. Evaluation Results

Figure 3 depicts the resulting lieftime improvement and wear-leveling potential for the aforementioned configurations. All illustrated numbers are the computed improvement factor in comparison to the baseline (static and unmodified mapping). Thus, a number larger than one indicates an improvement, a number smaller than one indicates a diminishment. The left subfigure each illustrates the computed lifetime improvement, the right subfigure the wear-leveling potential improvement respectively. The results are grouped by the operating wear-leveling strategy, every bar represents a tree and input data configuration. For the big tree configurations, we only include linear insert patterns with 50% insert distribution and also only OCTO$^+$ and AA configurations due to limited simulation time.

It can be observed that the improvement in terms of memory lifetime strongly depends on the wear-leveling strategy and input data configuration. It can be reported that both, the random and the ring strategy, decrease the memory lifetime significantly by at last 45%. This supports our assumption, that additional wear-leveling actions have to be performed very careful and very seldom. Blindly remapping memory blocks during checkpoiting induces the risk of applying heavy writes to a memory location, which may not have been modified anyway. Furthermore, it can be seen that for the input data configurations with linear data, we achieve up to 3× lifetime improvement with inter block wear-leveling only. It can be

observed, that solely applying inter block wear-leveling or combining inter and intra block wear-leveling turns out to be a decision, which is dependent on the input data. For instance for linear input data, additional intra block wear-leveling does not gain further improvement. Contrarily, for random input data, applying additional intra block wear-leveling increases the lifetime improvement further up to a factor of $1.69\times$ for small trees. For the YCSB input data, performing inter block wear-leveling solely or inter and intra block wear-leveling combined reports to not cause a huge difference. The potential for additional global wear-leveling on the granularity of 4096 byte memory pages (WLP) is only decreased in one case, compared to a static and unmodified mapping. Indeed combined inter and intra block wear-leveling improves the potential by a considerable factor of up to $1.46\times$, which possibly further increases memory lifetime when a global wear-leveling scheme is employed. In general, the results for small and big trees show a consistent result.

VII. Conclusion and Outlook

In this paper, we propose an orthogonal extension to generic application transparent in memory wear-leveling for non-volatile main memories. We intentionally establish an interplay of a B^+ tree as the application and the wear-leveling subsystem by hooking into checkpointing of the B^+ tree and exploit the mapping between the tree and the checkpoint to perform a lightweight wear-leveling. A precise bitwise evaluation reports up to $3\times$ extended memory lifetime while the memory space is even further groomed to improve coarse-grained generic wear-leveling.

In future work, we plan to enhance the evaluation to also consider metadata information, which are required for the checkpointing but are not considered in this paper. Furthermore, we plan to investigate NVM optimized B^+ trees from the literature. Applying the B^+ tree wear-leveling not only during checkpointing can be also considered.

Acknowledgements

This paper has been supported by Deutsche Forschungsgemeinshaft (DFG), as part of the project OneMemory (405422836) and SFB 876, subprojects A1 and A2 (http://sfb876.tu-dortmund.de/).

References

[1] H. Aghaei Khouzani, Y. Xue, C. Yang, and A. Pandurangi. 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*, pages 327–330, 2014.

[2] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[4] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao. Emerging nvm: A survey on architectural integration and research challenges. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(2):1–32.

[5] Y.-M. Chang, P.-C. Hsiu, Y.-H. Chang, C.-H. Chen, T.-W. Kuo, and C.-Y. M. Wang. Improving pcm endurance with a constant-cost wear leveling design. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(1):1–27, 2016.

[6] 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. In *Proceedings of the 49th Annual Design Automation Conference*, pages 453–458.

[7] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[8] P. Chi, W.-C. Lee, and Y. Xie. Adapting b+-tree for emerging nonvolatile memory-based main memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1461–1474, 2015.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[10] J. Dong, L. Zhang, Y. Han, Y. Wang, and X. Li. Wear rate leveling: Lifetime enhancement of pram with endurance variation. In *Proceedings of the 48th Design Automation Conference*, pages 972–977, 2011.

[11] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé. Increasing pcm main memory lifetime. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*.

[12] V. Gogte, W. Wang, S. Diestelhorst, A. Kolli, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. Software wear management for persistent memories. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*.

[13] C. Hakert, K.-H. Chen, and J.-J. Chen. Can wear-aware memory allocation be intelligent? In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 83–88, 2020.

[14] C. Hakert, K.-H. Chen, P. R. Genssler, G. von der Brüggen, L. Bauer, H. Amrouch, J.-J. Chen, and J. Henkel. Softwear: Software-only in-memory wear-leveling for non-volatile main memory. *arXiv preprint*.

[15] C. Hakert, K.-H. Chen, S. Kuenzer, S. Santhanam, S.-H. Chen, Y.-H. Chang, F. Huici, and J.-J. Chen. Split'n trace nvm: Leveraging library oses for semantic memory tracing. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2020.

[16] 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 *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 651–658. IEEE, 2020.

[17] K. Huang, Y. Mei, and L. Huang. Quail: Using nvm write monitor to enable transparent wear-leveling. *Journal of Systems Architecture*.

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

[19] W. Li, Z. Shuai, C. J. Xue, M. Yuan, and Q. Li. A wear leveling aware memory allocator for both stack and heap management in pcm-based main memory systems. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 228–233. IEEE, 2019.

[20] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386, 2016.

[21] 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*, 14(2):140–143, 2015.

[22] 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. *ACM SIGARCH Computer Architecture News*.

[23] 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 *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*.

[24] C. Wang and S. Chattopadhyay. Isle-tree: A b+-tree with intra-cache line sorted leaves for non-volatile memory. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 573–580. IEEE, 2020.

[25] J. Yang, Q. Wei, C. Wang, C. Chen, K. L. Yong, and B. He. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. *IEEE Transactions on Computers*, 65(7):2169–2183, 2015.

[26] 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. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2015.

[27] 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*, 37(3):14–23, 2009.

[28] L. Zhu, Z. Chen, F. Liu, and N. Xiao. Wear leveling for non-volatile memory: A runtime system approach. *IEEE Access*.