A Wear-Leveling-Aware Fine-Grained Allocator for Non-Volatile Memory

Xianzhang Chen[†], Zhuge Qingfeng[§], Qiang Sun[†], Edwin H.-M. Sha[§], Shouzhen Gu[§], Chaoshu Yang[†], Chun Jason Xue[‡]
[†]Chongqing University, Chongqing, China
[§]East China Normal University, Shanghai, China
[‡]City University of Hong Kong, Hongkong, China
{xzchen109, qfzhuge, edwinsha, yangchaoshu}@gmail.com, sqnaven@hotmail.com, szgu@sei.ecnu.edu.cn, jasonxue@cityu.edu.hk

ABSTRACT

Emerging non-volatile memories (NVMs) are promising main memory for their advanced characteristics. However, the low endurance of NVM cells makes them vulnerable to frequent fine-grained updates. This paper proposes a Wear-leveling Aware Fine-grained Allocator (WAFA) for NVM. WAFA divides pages into basic memory units to support fine-grained updates. WAFA allocates the basic memory units of a page in a rotational manner to distribute fine-grained updates evenly on memory cells. The fragmented basic memory units of each page caused by the memory allocation and deallocation operations are reorganized by reform operation. We implement WAFA in Linux kernel 4.4.4. Experimental results show that WAFA can reduce 81.1% and 40.1% of the total writes of pages over NVMalloc and nvm_alloc, the state-of-the-art wear-conscious allocator for NVM. Meanwhile, WAFA shows 48.6% and 42.3% performance improvement over NVMalloc and nvm_alloc, respectively.

1 INTRODUCTION

Non-volatile memories (NVMs), e.g. 3D XPoint [1] and Phase Change Memory (PCM) [2], are expected to be a mainstream memory device for their advanced characteristics. The application of NVM, however, needs to consider wear leveling for NVM has low endurance [3, 4]. As a fundamental data structure of memory management, memory allocator is a critical part for supporting wear-leveling [5–7]. Memory allocator is expected to provide less wear memory pages for distributing writes on the memory cells evenly.

Nevertheless, most of existing NVM allocators ignore the wear-leveling problem of NVM cells, especially for the wear leveling of memory cells within a page. The frequent fine-grained data updates of CPS applications and mobile systems [8] make the wear-leveling problem of NVM even worse. First, the page-grained allocators [9] may converge writes on minority memory cells in the beginning of a page, leaving the rest memory cells of the page untouched. Second, most of existing fine-grained memory allocators [10, 11] focus on improving space utilization rather than wear-leveling. Even though NVMalloc [11] sets a timing period to prevent continually memory allocation of a same page, it still suffers from imbalanced wear of memory cells in the same page.

In the aim of supporting wear leveling, this paper proposes a Wear-leveling Aware Fine-grained Allocator (WAFA) for the space management of NVM. First, WAFA divides a memory page into multiple memory units. Once an NVM page is taken apart, the information of the units of the page is persisted in the last unit for failure recovery. Second, we design a *Clockwise Best-Fit* (CBF) strategy to allocate the units of each page in a rotational manner, through which the memory cells in a page can be uniformly worn as much as possible. WAFA groups the pages in multiple buckets according to the maximum consecutive free units in each page. In the proposed CBF strategy, WAFA first selects a page from the best-fit bucket. Then, WAFA allocates free memory units of the page following the clockwise order. Finally, WAFA finds out the maximum consecutive free memory units in the page in a clockwise manner and moves the page to the proper bucket.

We implement WAFA in Linux kernel 4.4.4. WAFA is evaluated with both basic workloads and application workloads. We adopt workloads of Memcached [12] and YCSB [13] to represent real application workloads. We compare WAFA with NVMalloc [11] and nvm_alloc [14], the state-of-the-art wear-aware NVM allocator. The experimental results show that the total write counts of pages in WAFA achieves 81.1% and 40.1% reduction comparing with NVMalloc and nvm_alloc. These results indicate that WAFA can significantly improve the wear-leveling of NVM page. Furthermore, WAFA also shows 48.6% and 42.3% performance improvement over NVMalloc and nvm_malloc, respectively.

The main contribution of this paper includes:

- We propose a novel design of wear-leveling aware NVM management scheme and a memory allocation algorithm *CBF* for fine-grained memory requests.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317752

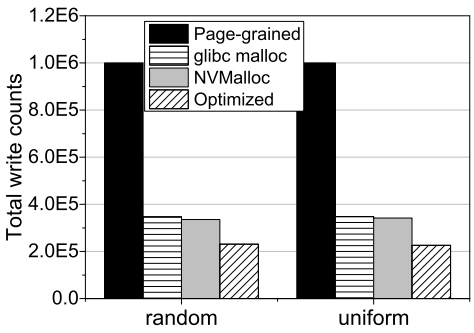


Figure 1: Motivational example.

- We propose an efficient mechanism to reform the free memory units of divided pages.
- Extensive experimental results show that the proposed approach achieves significant improvement for wear-leveling and performance over existing approaches.

The rest of this paper is organized as follows. Section 2 introduces the background and shows the motivational example. Section 3 presents the design of the proposed WAFA allocator for NVM. Section 4 compares the proposed WAFA with existing memory allocators by extensive experiments. Finally, Section 5 concludes the paper.

2 BACKGROUND AND MOTIVATION

CPS applications show two critical characteristics in data requests. First, the new data in updating requests are generally fine-grained, i.e., their sizes are less than the size of a common memory page (i.e., 4KB). For example, the smart home traces, Smart* Data Set obtained from UMass Trace Repository [15], show that the data detected by sensors in each period can be stored in a space less than 128 bytes or even less. Therefore, it is necessary to provide fine-grained space management for storing small data. Second, the data store of these applications constantly writes new data to a free space rather than update it in an in-use place. For example, the commonly-used key-value stores (e.g., Memcached [12]) always write new key-value pairs in free space. Therefore, the space allocator needs to allocate less worn space for new data requests. Based on the two observations, we believe that it is necessary to design a wear-leveling aware fine-grained allocator for NVM.

Existing fine-grained memory allocator can be classified as two types according to their design objective. One type is designed for managing conventional DRAM. This type of allocators, such as *glibc malloc*, *jemalloc* [16], and *Hoard* [17], have no wear consideration, thus, they are not suitable for NVM management. For example, glibc malloc and its variants allocate space in smallest first and best-fit way. They cache and reuse small sub-pages to boost allocation performance. Besides, the metadata is updated for any allocation/deallocation request, resulting in excessive writes to metadata pages.

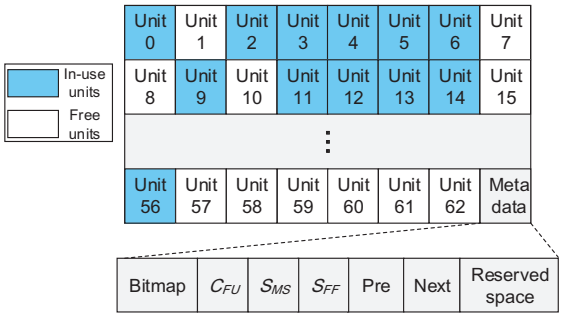


Figure 2: Illustration of a divided page.

The other type is designed for managing the upcoming NVM. Most of existing NVM allocators [7, 9, 10, 14], such as nvm_alloc [14] and Makalu [10], do not consider wear-leveling for fine-grained updates, except for WAlloc [18] and NVMalloc [11]. WAlloc [18] organizes NVM as small areas that each area has two wear counters for recording the write counts of the area. In allocation, WAlloc uses a Less Allocated First Out (LAFO) strategy to allocate the memory areas with a wear count that is smaller than the average wear count. To use LAFO strategy, WAlloc need to compute the average write counts of memory areas and adjust the priority of memory areas according to their write counts. Thus, WAlloc has large performance and space overhead for achieving wear-leveling of fine-grained write request. NVMalloc [11] sets a period called "don't-allocate time" for each memory page. A memory page is allowed to be written at most twice in a period. However, the wear-leveling of sub-pages is ignored. Hence, NVMalloc may heavily wear a page by repeatedly using minority sub-pages.

We show a motivational example of optimizing the in-page wear-leveling by measuring the total wear counts of pages using different allocators. In the experiments, we conduct one million randomly and uniformly generated small update requests on each allocator. We use the maximum write counts of the memory cells in a page as the wear count of the page. The experimental results are shown in Figure 1. Compared with the page-grained allocator, the optimized allocator can reduce more than 76.9% and 77.4% wear counts for random and uniform workloads, respectively. Compared with NVMalloc, the optimized allocator can reduce more than 31.1% and 33.9% wear counts for random and uniform workloads, respectively. It is because that the optimized allocator can better balance the usage of the memory cells in each page. Therefore, we try to optimize the management of fine-grained memory space to achieve better wear-leveling support in allocator.

3 THE PROPOSED WAFA DESIGN

3.1 Overview

In the aim of providing an efficient wear-leveling allocator for fine-grained memory requests, the main idea of WAFA is to break a page into smaller parts and alternately allocate

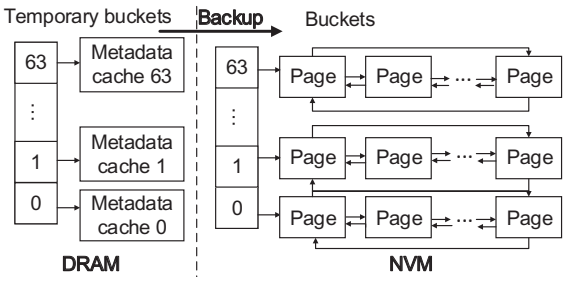


Figure 3: Illustration of the buckets of pages.

the units for new requests. Based on the idea, we propose the design of page metadata, a Best-fit Clock Strategy for allocation, and.

Basically, WAFA divides the memory pages into multiple units with the minimum allocation size. The minimum allocation size is the same with the size of the last-level cache line for smaller sub-page will cause write amplification by disturbing other sub-page's bits. The size of a unit is set to 64 bytes in this paper. Thus, each 4096B memory page is divided into 64 units, as shown in Figure 2. The last unit is used for storing the information of the units in the page, namely, "page metadata". Note that only a divided page contains page metadata that are initialized when the page is first used by WAFA. An unused page has no WAFA metadata.

The structure of a page metadata is shown in Figure 2. The beginning 8B bitmap indicates the status of the units. C_{FU} (one byte) means the total counts of free units in the page. In WAFA, the units in a divided page is regarded as "scales in a clock". WAFA allocates the best-fit units in a clockwise order. A cycle of this allocation process is called a *run*. We use S_{FF} (one byte) to record the current "scale", which is the ID of the first free unit in the rest of the run. For example, suppose unit 1 in Figure 2 is allocated in the previous memory request, then, S_{FF} is set to 7.

The consecutive free units in the rest of the run is called *segment*. We set S_{MS} (one byte) to store the size of the maximum segment indicated by S_{FF}. The size of S_{MS} ranges from 0 to 63, where size 0 and 63 means the page is fully in-use and free, respectively. For the page in Figure 2, suppose the current segment is unit 57-62, we have $S_{MS} = 6$.

In order to fast locate a page that satisfies the requested size, the pages are partitioned to 64 buckets according to the size of the current consecutive free units, as shown in Figure 3. The buckets are kept on a fixed NVM page for indexing the divided pages and free intact pages. The *Pre* and *Next* are two pointers (8 bytes each) for connecting the pages in the corresponding bucket.

To avoid heavy writes on the metadata of NVM pages, we construct a set of temporary buckets for caching the page metadata in DRAM during runtime. Each temporary bucket points to a metadata cache. For each page of the bucket on NVM, we maintain a copy of its metadata in the metadata cache of the corresponding temporary bucket. Concretely, for each page metadata, we only need to store its bitmap, C_{FU}, S_{MS} and S_{FF},*Pre*, and *Next* in the metadata cache. Thus, each 4KB page in the WAFA allocator needs 27B in DRAM. The DRAM space overhead of WAFA is about 0.66% of the total size of the NVM allocator.

The available free units may be scattered in the whole page after extensive allocation/free operations. In order to improve space utilization and allocation efficiency, it is necessary to reform the consecutive units of each divided page and migrate the divided page to proper bucket. WAFA backs up the metadata cache of a page to the corresponding page metadata each time the page is reformed. In normal system shutdown, the allocator can be easily recovered using the persisted bucket array and page metadata. For the case of system failure, the system reconstruct the WAFA allocator by scanning the memory. Therefore, we do not consider further consistency issues in the design of WAFA.

3.2 Clockwise Best-Fit Allocation

To balance the wear of memory cells in a page, a first approach could be using a wear counter for each unit in the page and always pick the least worn unit for allocation. However, this approach will cause extremely large timing cost for updating and querying wear counters and large space overhead (e.g., 12.5%=8B/64B*100%) for storing wear counters.

In this paper, we propose a *Clockwise Best-Fit* (CBF) strategy to allocate units for fine-grained memory requests. The main idea of CBF is twofold. First, we allocate best-fit consecutive free units for memory allocation request, through which the memory cells have highest chance to be written after allocation. Second, the units are allocated in a clockwise order. The detailed allocation steps using CBF is shown in Algorithm 1.

Algorithm 1: Clockwise Best-Fit allocation.

Input: S_r: the size of requested new memory space; S_u: the size of a memory unit;
Output: The ID A_r of selected unit for allocation;

1 $N_r \leftarrow \lceil S_r/S_u \rceil$;
2 **while** $Bucket_{N_r}$ *is empty* **do**
3 $\quad N_r++$;
4 $A_r \leftarrow Pos(P_i)$ of the first page P_i in $Bucket_{N_r}$;
5 $C_{FU}(P_i) \leftarrow C_{FU}(P_i) - N_r$;
6 $S_{MS}(P_i) \leftarrow$ get the maximum segment in the rest of the run;
7 $Pos(P_i) \leftarrow$ ID of the first unit in $S_{MS}(P_i)$;
8 Move page P_i to the tail of $Bucket_{S_{MS}(P_i)}$;

In the algorithm, we first calculate the necessary size of segment for allocation request. Then, WAFA tries to allocate required segment from the best-fit bucket. CBF obtains a free segment from a larger bucket when the current bucket is null. WAFA sets the ID of the allocated memory space, namely A_r, to the ID $Pos(P_i)$ the first memory unit of the selected segment and updates the total number of free units

in age P_i. In line 6-7, if the maximum segment is fully used (i.e. $S_{MS}(P_i) = 0$) and the page still has free unit(s), WAFA searches for the maximum segment in the rest of the run and updates corresponding metadata. Finally, the page is moved to the tail of $Bucket_{S_{MS}(P_i)}$. After serving allocation requests, the maximum segment in a page may change. Thus, the affected page should be moved to a new bucket. Therefore, it is necessary to fast determine the maximum segment. Intuitively, we can scan all the units and count one by one, but it cause large timing overhead.

The main complexity of CBF comes from step 2 and step 6. The timing complexity of step 2 is $O(|Buckets|)$. In step 6, to get the maximum segment in the rest of the run, we adopt a divide-and-conquer based approach to calculate the free bits in the bitmap of the basic units. The algorithm can find out the maximum segment in $O(n)$ time by caching the status of each part of the bitmap. As a result, the timing complexity of CBF is $O(|Buckets| + n)$.

In implementation, we only need 64 buckets. The 8-byte bitmap is divided into four 2-byte slices. Each 2-byte slice can be regarded as an integer range from 0 to 65535. We construct an auxiliary array to store all the 65536 statuses of the slices, including the sizes of three parts of each slice. The size of each part can be represented by at most 4 bits for the length of a slice is 16 bits. The array is placed on NVM for its read-only. Thus, the space overhead of auxiliary array is 96KB which is negligible for memory management. The allocation procedure needs to consult the metadata to determine the proper size and bucket. WAFA accelerates the allocation procedure by using the metadata cache in DRAM. After a successful allocation, the metadata cache is flushed to the corresponding NVM page to reflect the changes made due to the last allocation.

3.3 Free Operation and Reforming Operation

Here, we introduce the free operation of WAFA and the reforming of divided pages. For a free operation, WAFA simply updates the page metadata information related to the freed memory units in metadata cache. Concretely, a free operation is accomplished as follows:

(1) The allocator finds out the related page metadata by calculating the address of freed units.
(2) The allocator resets the related bits of the bitmap in the corresponding metadata cache.
(3) The allocator updates C_{FU} in the metadata cache.

After executing lots of allocation and free operations, the divided pages in the system show following states:

- Many pages have completed a run and regain lots of free units via free operations.
- Many pages may contain lots of free units but stuck in the last several units of the run.
- The maximum segment in a run may be enlarged by free operations but S_{MS} still stores the old value.

Thus, we propose reforming operation to unite the memory units in the divided pages and adjust the divided pages to proper buckets.

A reform operation is responsible for calculating the maximum segment of divided pages and backs up the metadata cache in DRAM to the corresponding page metadata in NVM. Even though reforming operation is important for regaining free memory units, we want to minimize the impact of reforming on the performance of allocation and free operations. Let NP_k be the total number of pages in bucket k, The average number of free units $TF_k.avg$ of bucket k is:

$$TF_k.avg = \frac{\sum_{q=1}^{NP_k} C_{FU}(P_q)}{NP_k}$$

As aforementioned, the reforming operation is stimulated by free operations. Thus, we commit a reforming operation on a bucket if and only if the average number of free units of the pages in the bucket exceeds the preset segment size of the bucket, i.e. $TF_k.avg > k$.

In implementation, WAFA adopts a daemon thread to perform reforming operation if necessary. A reforming operation is accomplished as follows:

(1) The daemon thread calculates $TF_k.avg$ of bucket k.
(2) If $TF_k.avg > k$, for each page in bucket k, the thread calculates the maximum segment of the page. The S_{MS} and Pos of the pages are reset according the maximum segments.
(3) Then, WAFA adds the reformed pages to temporary buckets according to its S_{MS}.
(4) The thread merges each temporary bucket to the bucket of allocator with the same S_{MS}.

As a fined-grained allocator, WAFA can be used together with a page-level allocator to achieve system level wear leveling. In this case, we can limit the number of available pages in WAFA in case of memory shortage caused by fragments. The metadata cache is backed up to the corresponding page metadata in NVM after reforming. The complex structure changes require widely-used consistency techniques [10] to avoid permanent corruption of NVM data.

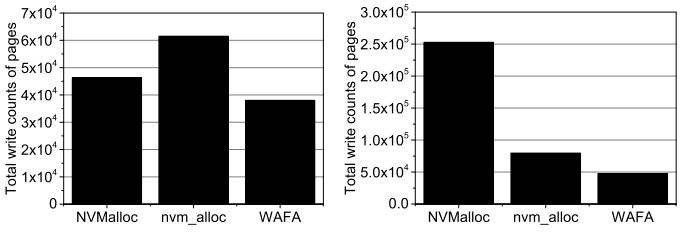
4 EVALUATION

4.1 Experimental Setup

We implement WAFA in Linux kernel 4.4.4 with dedicated APIs. WAFA is compared with *glibc malloc* (denoted by malloc), *nvm_alloc* [14], and *NVMalloc* [11], the state-of-the-art wear-conscious NVM allocator. NVMalloc sets a timing period for each page and ensures that a page can be written at most twice during the timing period. The performance of NVMAlloc highly depends on the timing period. The same as the paper of NVMalloc [11], we set the timing period to 100ms in the experiments.

The four allocators are evaluated with two types of workloads. The Memcached [12] workload is composed of 60K inserts and 40K random deletes that each operation relates to a 10B key and 256B value. In YCSB [13] workloads, we

configure four smart home scenarios and generate one thousand record for each scenario by YCSB. The sizes of the records range from 4B to 32B. Then, we repeatedly store and destroy the mixed four scenarios for one million times.



(a) Memcached workloads (b) YCSB workload

Figure 4: Comparison of the total write counts of pages using different workloads.

The experiments are conducted on a machine equipped with an Intel i3-3220 processor and 8GB DDR3 DRAM. We configure 4GB memory to stand for NVM. In the experiments, we record the allocation counts of each minimum memory unit (64B in the experiments) in a page. Since a page is destroyed if one of its memory cell reaches the endurance limitation [2], we use the maximum allocation counts of the units in a page as the wear counts of the page. The experimental results show that WAFA gains significant improvement for wear-leveling over glibc malloc and NVMalloc.

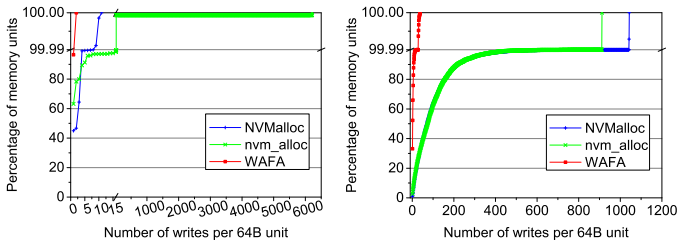
4.2 Effect on Wear leveling

Here, we evaluate the wear-leveling effect of the algorithms by evaluating the write counts of pages and the write count distribution of memory units.

4.2.1 Total write counts of pages. From the perspective of system software, the write count of a page is the maximum write counts of the memory cells within the page. In the experiments, we calculate the write count of every divided page and use the sum of the write counts of all the divided pages as total write counts. Less total write counts of pages means better wear-leveling.

Figure 4 (a) shows the experimental results evaluated by the Memcached workloads [12]. WAFA achieves least total write counts of pages. Compared with NVMalloc and nvm_alloc, WAFA reduces 17.9% and 38.2% of the total write counts, respectively. The memory requests in Memcached workload is fixed. Each insert operation and delete operation contains a pair of two memory request: the key relates to four consecutive memory units and the value relates to one memory unit. As a consequence, WAFA can distribute the writes evenly on memory units in a page using the proposed CBF strategy. On the contrary, NVMalloc and nvm_alloc are unaware of the status of the memory units in the pages and frequently reuse seldom memory units.

Figure 4 (b) shows the experimental results evaluated with the key-value store workloads generated by YCSB [13]. WAFA shows 81.1% and 40.1% improvement over NVMalloc and nvm_alloc, respectively. The experimental results indicate that WAFA can better support wear-leveling for small-sized data requests. The memory requests in the YCSB workloads all range from 4B to 32B, i.e., just one memory unit. Therefore, the proposed CBF strategy in WAFA can allocate every free memory unit of a page by turns, through which the writes are evenly distributed to the free memory units in the run. NVMalloc uses a "don't-alloc list" to prevent that a memory unit is written twice during a timing period, 100ms in the experiments. NVMalloc is unaware of the write distribution of the memory units in a page. Thus, a heavy written memory unit can be reused by skipping a period.



(a) Memcached workloads (b) YCSB workloads

Figure 5: CDF of writes per memory unit in Memcached and YCSB workloads. The top of the figures expands the 99.99%-100% interval.

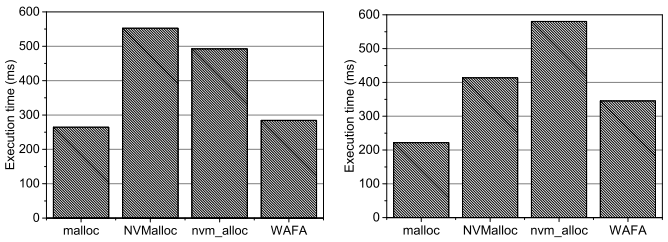
4.2.2 Write distribution of memory units. Here, we evaluate the write count distribution of the 64B memory units. Figure 5 shows the CDFs of writes per memory unit in the workloads. The X axis shows the write counts of per 64B memory unit. We set the free ratio to 80% for basic workloads. WAFA achieves significant improvement over NVMalloc and nvm_alloc for both basic and YCSB workloads.

Figure 5 (a) shows the results of Memcached workloads. In the Memcached workloads, the memory units in WAFA and NVMalloc are all written less than 5 times and 20 times, respectively. The write counts of nvm_alloc are highly imbalanced for 99% of the memory units in nvm_alloc are written less than 24 times and four memory units are written 6211 times. nvm_alloc sets a memory unit as the header for each types of memory space. These headers may be heavily written for nvm_alloc is unaware of wear leveling. Thus, the NVM device nvm_alloc can be easily damaged using the Memcached workloads. The standard deviation of the write counts of memory units in NVMalloc, nvm_alloc, and WAFA is 1.41, 15.9, and 0.182, respectively. The CV of WAFA is 69.4% and 97.3% lower than that of NVMalloc and nvm_alloc, respectively. WAFA still gains the best wear leveling of the memory units.

The results of YCSB workloads in Figure 5 (b) show similar trends. The write counts of the memory units in NVMalloc, nvm_alloc, and WAFA is 1 to 1043, 1 to 912, and 1 to 40, respectively. The standard deviation of the write counts of memory units in NVMalloc, nvm_alloc, and WAFA is 101.1, 97.5, and 2.93, respectively. The CV of WAFA is 13.6% and

11% lower than that of NVMalloc and nvm_alloc, respectively. In conclusion, WAFA achieves better wear-leveling than the state-of-the-art NVM management approaches.

The write distributions of memory units help explain the results of total number of writes of pages. For example, although there is only one memory unit written 1043 times by NVMalloc, the system software still regards the write count of the page as 1043 even if the rest of memory units in the page are never written. Therefore, it is necessary to achieve in-page wear-leveling especially for the applications with lots of fine-grained memory requests.



(a) Memcached workloads **(b) YCSB workloads**

Figure 6: Comparison of execution time of different workloads.

4.3 Effect on Performance

Next, we evaluate the performance of WAFA. We use malloc as the upper bound for it has no wear-leveling and consistency mechanisms. The experimental results are shown in Figure 6. WAFA outperforms NVMalloc and nvm_alloc for all cases.

Figure 6(a) shows the experimental results on Memcached workloads. WAFA still outperforms NVMalloc and nvm_alloc. The free ratio of Memcached workload is 40% for it commits 60K inserts and 40K inserts. The execution time of WAFA is 7.4% more than that of malloc for WAFA provides wear-leveling techniques. Compared with NVMalloc and nvm_alloc, WAFA saves 48.6% and 42.3% execution time, respectively.

Figure 6(b) shows the experimental results of using YCSB workloads. WAFA still shows higher performance than NVMalloc and nvm_alloc. Compared with NVMalloc and nvm_alloc, the performance improvement of WAFA is 16.5% and 40.5%, respectively. In this case, the memory requests in the YCSB workloads are relatively regular for the size only ranges from 4B to 32B, i.e., less than just one memory unit. In nvm_alloc, the totally small sized memory requests cause large overhead for the frequent chunk and block breakdowns in the three-level memory allocation process.

5 CONCLUSION

In this paper, we studied the wear-leveling problem of NVM confronting fine-grained memory allocation requests. We proposed WAFA to solve the problem. In WAFA, the pages are divided into basic units and allocated in rotation for allocation requests. The basic units are alternatively used for reducing the wear of pages caused by small writes. We implemented WAFA in Linux kernel for evaluation. The experimental results verified that WAFA can achieve significant total wear counts reduction over existing memory allocators.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable comments and improvements to this paper. This work is partially supported by grants from the National Natural Science Foundation of China (No. 61472052 and No. 61802038), the Chongqing Postdoctoral Special Science Foundation (No. XmT2018003), the China Scholarship Council (No. 201706050116 and No. 201706050117), and the China Postdoctoral Science Foundation (No. 2017M620412).

REFERENCES

[1] I. Corporation, https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/, 2015.
[2] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA*, 2009.
[3] F. Huang, D. Feng, Y. Hua, and W. Zhou, "A wear-leveling-aware counter mode for data encryption in non-volatile memories," in *DATE*, 2017.
[4] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes," in *MICRO*, 2018.
[5] J. Hu, Z. Zhuge, J. C. Xue, W.-C. Tseng, and E. Sha, "Software enabled wear-leveling for hybrid pcm main memory on embedded systems," in *DATE*, 2013.
[6] Q. Li, Y. He, Y. Chen, J. C. Xue, N. Jiang, and C. Xu, "A wear-leveling-aware dynamic stack for pcm memory in embedded systems," in *DATE*, 2014.
[7] W. Li, Z. Shuai, J. C. 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 *DATE*, 2018.
[8] Y.-H. Kuan, Y.-H. Chang, T.-Y. Chen, P.-C. Huang, and K.-Y. Lam, "Space-efficient index scheme for pcm-based multiversion databases in cyber-physical systems," *ACM TECS.*, vol. 16, no. 1, pp. 21:1–21:26, 2016.
[9] H. A. Khouzani, C. Yang, and F. S. Hosseini, "Segment and conflict aware page allocation and migration in dram-pcm hybrid main memory," *IEEE TCAD*, vol. 36, no. 9, pp. 1458 – 1470, 2016.
[10] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *ACM OOPSLA*, 2016.
[11] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *ACM TRIOS*, 2013, pp. 1:1–1:17.
[12] "Memcached: A distributed memory object caching system," http://www.memcached.org/, 2018.
[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.
[14] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm_alloc: Memory allocation for nvram," in *AWMD*, ser. in conjunction with VLDB, 2015.
[15] "Umass trace repository. smart* data set for sustainability," http://traces.cs.umass.edu/index.php/Smart/Smart, 2017.
[16] J. Evans, "A scalable concurrent malloc(3) implementation for freebsd," http://www.bsdcan.org/2006/papers/jemalloc.pdf, 2006.
[17] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *ASPLOS IX*, 2000.
[18] S. Yu, N. Xiao, M. Deng, F. Liu, and W. Chen, "Redesign the memory allocator for non-volatile main memory," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 49:1–49:26, Apr. 2017.