A Wear Leveling Aware Memory Allocator for Both Stack and Heap Management in PCM-based Main Memory Systems

Wei Li*, Ziqi Shuai*, Chun Jason Xue†, Mengting Yuan*, ✉Qingan Li*
*School of Computer Science, Wuhan University, China
†Department of Computer Science, City University of Hong Kong, Hong Kong
{weili, szq}@whu.edu.cn, jasonxue@cityu.edu.hk, {ymt, qingan}@whu.edu.cn

***Abstract*—Phase change memory (PCM) has been considered as a replacement of DRAM, due to its potentials in high storage density and low leakage power. However, the limited write endurance presents critical challenges. Various wear leveling techniques have been proposed to mitigate this issue from different perspectives, including both hardware and software levels. This paper proposes a wear leveling aware memory allocator, which (1) always prefers allocating memory blocks with less writes upon memory requests, and (2) leaves blocks allocated more than a threshold value unallocable temporarily. Furthermore, for the first time, this allocator provides a uniform management scheme for both stack and heap areas, thus could better balance writes in stack and heap areas. Experimental evaluations show that, compared to state-of-the-art memory allocators (i.e., *glibc malloc*, NVMalloc and Walloc), the proposed memory allocator improves the PCM wear leveling, in terms of CoV (a wear leveling indicator) by 41.9%, 30.3%, and 35.8%, respectively.**

***Index Terms*—PCM, wear leveling, memory allocator, stack and heap**

I. INTRODUCTION

Dynamic Random-Access Memory (DRAM) has achieved dramatic density improvement along with the advancements of memory technology scaling over the past few decades. However, continued scaling will be in jeopardy and scaling DRAM beyond 22nm seems impossible [1], which restrict the amount of memory available to a system. Moreover, in order to prevent the data loss caused by the leakage of electric charge on capacitors, DRAM consumes significant energy to continuously refresh power, contributing to as much as 20-40% of total system energy [2].

Fortunately, the emerging Phase Change Memory (PCM), one of non-volatile memory (NVM) technologies, has demonstrated great potentials to be an alternative to DRAM due to their high integration density, low leakage power and byte addressability [3] [4] . However, the limited write endurance of PCM presents challenges for integrating it in the memory hierarchy. PCM cells will be destroyed permanently after enduring 10^8 writes, causing their average lifetime less than three years. Furthermore, the write-intensive operations will accelerate the wear of PCM and shorten the achievable lifetime to even several days [5].

To alleviate the endurance problem and extend lifetime of PCM, lots of wear leveling techniques based on operating system (OS) technology and compilation technology were proposed, striving to evenly distribute write operations among the entire PCM space. Specifically, majority of OS level techniques focus on achieving page level wear leveling by swapping hot and cold pages [6] [7]. The compilation level techniques, on the other hand, are mainly engaged in reducing uneven wear caused by data writing patterns, and achieve program memory level wear leveling by optimizing the data allocation scheme [8] [9], which are somehow orthogonal to OS level techniques. In this study, we focus on the latter.

It is observed that the mechanism of stack and heap allocation is one of the major causes of uneven data writes during program's runtime. Some existing work proposed wear leveling aware stack allocation or heap allocation respectively [9] [10] [5]. However, these methods either consider stack or heap areas, but never both. We conducted a series of experimental evaluations of the usage of PCM cells, and observed that there is a serious imbalance between the wear of stack and heap as well. Based on this observation, this paper proposes to employ a wear leveling aware memory allocator for both stack and heap management. The major contributions of this paper are highlighted as below:

- To the best of our knowledge this paper presents the first method that uniformly managing the stack and heap area to achieve wear leveling inside and between stack and heap areas.
- A wear leveling aware memory allocator based on wear count per basic block is proposed to achieve write balance over the memory space.

The rest of paper is organized as follows. Section II discusses the background and motivation. Section III introduces the uniform management for stack and heap. Section IV describes the implementation of our wear leveling aware memory allocator. Section V presents the experimental evaluation. The related work is discussed in Section VI. Finally, Section VII concludes this paper.

II. BACKGROUND AND MOTIVATION

A. PCM-based main memory system

a) Phase Change Memory (PCM): PCM is a type of non-volatile random-access memory that utilizes the large

resistivity contrast between crystalline and amorphous phases of chalcogenide (the phase-change material) to store bits, where the crystalline state (low resistivity) corresponds to 1 and the amorphous state (high resistivity) corresponds to 0. Phase changes are induced by injecting current and heating the memory cell, which determines the energy consumption of writes operations. Besides, the transformation from amorphous to crystalline takes a little longer time, which determines the write latency. Table I presents the parameters of PCM and DRAM [10] [5].

b) A PCM Main Memory Architecture: Based on the characteristics described above, we observe that PCM and DRAM each have their respective pros and cons. If substituting PCM for DRAM as main memory directly, the system performance will be decreased significantly because of the relatively higher latency and worse endurance of PCM compared to DRAM. Instead, a PCM main memory architecture usually comprises a small DRAM, harnessing the best properties of each. In this study, we assume that the operating system (OS) will manage the PCM in a manner similar to current DRAM systems. PCM will be organized at 4 KB page granularity, and processes can request PCM pages from OS by system call *mmap()*. DRAM will be utilized to absorb some frequent writes, which requires the virtual memory mechanism to be able to map both PCM and DRAM into process address space. Lastly, in order to take the advantage of PCM's non-volatile characteristic and restore persistent application state, OS must maintain the non-volatile memory pages for each process across machine reboots.

B. Uneven wear inside and between stack and heap

a) Uneven Wear inside Stack: The stack is utilized to manage the procedure calls and functions' execution context in program's runtime environment. Each function instance has its own unique associated memory on the process's stack, which is called stack frame or active record, to hold incoming parameters, local variables and temporary variables. The conventional stack based memory allocator works in last-in-first-out (LIFO) method, and the function's frame allocation/deallocation is always conducted on the top of stack. Therefore, some memory regions may be allocated to a large amount of frames, while some others are rarely used.

Reference [9] quantified the uneven wear problem of stack by conducting a set of experimental evaluations of the writing patterns on stack, and proposed a Wear Leveling aware Dynamic Stack (WLDS) technique to mitigate this problem. This method employs a dynamic stack allocator for the management of stack. The allocator works in a way analogous to the heap based memory allocator, which manages a free block list and allocates a free block from the list following a next-fit policy.

TABLE I: Characteristics of PCM and DRAM

	Read	**Write**	**Endurance**	**Write power**
DRAM	$30ns$	$15ns$	$> 10^{16}$	$0.1nJ/b$
PCM	20–50ns	60–120ns	10^8	$< 0.1nJ/b$

However, they don't deal with the write imbalance in heap area, and the next-fit policy based technique is suboptimal.

b) Uneven Wear inside Heap: Distinct from stack, heap is more explicitly managed and free-floating. Considering *glibc malloc*, one of the most universal heap based memory allocator at present, it caches and reallocates small memory blocks in last-in-first-out way, which may concentrate writes in a few memory areas. Additionally, malloc maintains metadata in the header and footer of each used/free memory block, containing the size of block and the pointers to the previous and next block. Metadata is updated whenever the block is merged or splitted, leading to a slew of small writes.

In order to reduce the uneven usage of heap memory, Reference [10] [5] respectively proposed a wear aware memory allocator as the substitute for the traditional allocator, namely NVMalloc and Walloc. Both of them leverage segregated free lists for the management of different size free blocks, and divert the frequently changing metadata (e.g. free lists) to DRAM. Furthermore, NVMalloc restricts on memory blocks so as to not be allocated more than once during the time interval T. To this end, the newly freed memory will be timestamped and inserted into a FIFO queue (the don't-allocate list). Walloc follows the Less-Allocated-First-Out (LAFO) policy and always allocates a free memory block with fewer writes. To avoid some free memory lists become hot under skew-size workloads, Walloc sets a global threshold to limit the free block lists' allocation frequency. Those free block lists, whose average wear count is beyond this threshold, will be unavailable for reallocation. However, these methods may fail in some applications, which we identify in section IV-D. Moreover, they do not deal with the imbalance in stack area.

c) Uneven Wear between Stack and Heap: Conventionally, stack and heap areas are managed separately and grow in opposite directions. This convention contributes to the different writes density between these two areas. As shown in Fig. 1, the data writing patterns between stack and heap may differ by orders of magnitude, where the maximum number of writes reaches almost 175 thousand in stack but less than 400 in heap.

Moreover, there is commonly a big gap between stack and heap areas to avoid early stack overflow or heap overflow. This gap, mostly untouched throughout a process's lifetime, significantly exacerbates the uneven wearing among the whole address space. As far as we know, none of existing compilation based techniques has been proposed to address this issue. For the first time, this paper presents a wear leveling memory allocator to manage both areas uniformly.

III. UNIFORM MANAGEMENT FOR STACK AND HEAP

The conventional stack and heap memory allocators are designed with the purpose of higher allocation speed and more efficient space utilization, which is somehow in conflict with wear leveling in current situation. These allocation mechanisms make sense for DRAM, but they will have unintended effect when applied to PCM. To tackle this problem, we

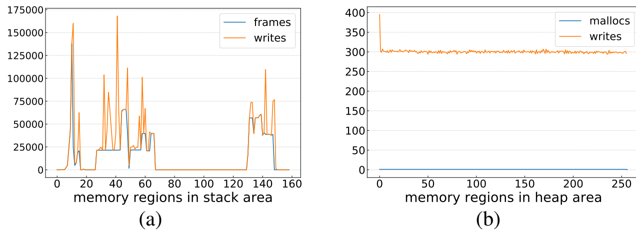


Fig. 1: Stack and heap memory usage of *basicmath* (a). the number of writes as well as function frames over stack area. (b). the number of writes as well as malloc requests over heap area.

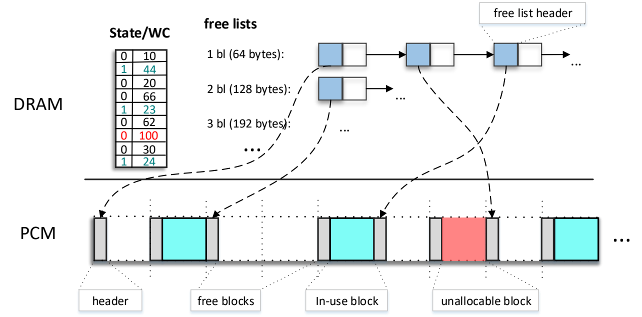


Fig. 2: Structure of UWLalloc

propose to employ a wear leveling aware allocator to allocate and manage memory for stack and heap uniformly, thus to achieve an even usage both inside and between stack and heap.

Applying a custom allocator for heap allocation is a fairly straightforward affair, since the built-in malloc implementation can be simply replaced for both dynamically linked and static linked programs. However, applying a custom allocator for stack allocation is relatively troublesome, as the traditional stack allocation is automatically completed, through adjusting of the stack pointer register. To address this issue, we adopt the dynamic stack proposed in [9], as stated below:

- Suppose that a caller function f is about to invoke a callee function g. Immediately before the invocation, the *malloc* function is employed to obtain an appropriate memory area for g's frame.
- An extra entry, control link(CL) is added into g's frame to store the frame address (the highest memory address) of caller function f.
- The register *ebp*'s content is updated with g's frame address, meanwhile, the local variables can be accessed through the negative offset to the frame address as usual.
- Then, function g executes afterwards.
- Lastly, upon function g returns, the *ebp*'s content is updated with CL value, pointing back to f's frame. And the *free* function is employed to release memory block used by g.

Under this circumstance, each function's frame can be regarded as an object, similarly to the object on the heap. Every time a function is invoked/finished, its frame object is allocated/released. Hence, the frame objects and heap objects are in a position to share an identical wear leveling aware allocator, and we can achieve a unified management of stack and heap, making no distinction between their memory space. As a result, the problem of uneven wear between stack and heap is considered as a whole, and it's no longer necessary to reserve a big gap of between stack and heap.

IV. WEAR LEVELING AWARE MEMORY ALLOCATOR

Traditional dynamic memory allocators (e.g. *glibc allocator*) couple the frequently changing metadata and data for easy used/free memory space management, and cache newly freed small memory blocks for preferential allocation. Thus heavily skew memory writes will arise when using these allocators. To address this issue, we proposed UWLalloc, a new Wear Leveling aware allocator for Uniform stack and heap management in PCM based main memory system. UWLalloc is based on the two ideas: (1) prefer allocating memory blocks with less writes upon memory requests, and (2) leave blocks allocated more than a threshold value unallocable temporarily.

A. Structure of UWLalloc

The structure of UWLalloc is similar to that of NVMalloc, shown in Fig. 2. Firstly, the segregated free list approach is adopted which maintains a set of linked lists, where each list holds free blocks of a particular size. Notice that the memory space allocated in UWLalloc is 64 bytes aligned, thus every memory block allocated is composed of basic memory blocks (64 bytes) and the size for every free list is a multiple of 64 bytes. Traditional allocators store the list pointers inside the headers and footers of freed blocks. Those in-place pointers are updated when coalescing contiguous free blocks or splitting large free block, resulting in a mass of small writes. To prevent PCM from these frequent small writes, UWLalloc relegates the list pointers to DRAM and links all the list nodes to their associated PCM memory area.

Secondly, a state list stored in DRAM is utilized to help track the allocated/free state and the wear count (WC) of each basic memory block. WC indicates the allocation frequency of memory block and determines the allocation order. Note that data in DRAM is volatile, and will be lost across reboots. To persist the allocation state, a header with its size and allocation state is coupled to each allocated/free memory area. When mapping a new PCM region, the volatile data can be rebuilt [10] except wear count. To restore wear count information, UWLalloc, like Walloc, checkpoints it to PCM at intervals.

B. Allocation algorithm

UWLalloc preferentially allocates memory blocks with the least allocation frequency for the purpose of wear leveling. Specifically, the memory blocks in free lists are ordered by worst wear count (the maximal wear count among the constituent basic memory blocks), therefore, we only need to consider the head block when picking one out from a free list. The steps of allocation are as follows:

- Upon a memory request, UWLalloc calculates the number of basic memory blocks needed for allocating, then selects the free list with the best fit size and returns the first memory block.
- If the corresponding list is empty, the lists with larger sizes will be searched, and a larger free block will be split.
- If none is found in all lists, a new memory area will be allocated from the free zone of requested memory, where the memory has never been allocated yet.
- On condition that the free zone memory space is insufficient and allocation still fails, more PCM pages will be requested from the operating system. In addition, the maximal memory space can be configured, and when memory usage approaches this ceiling, the wear limit will increased to recover those unallocable blocks
- At last, each allocated basic block's WC is increased by 1 and state is shifted from free to allocated.

C. Free algorithm

The simple less-allocated-first-out policy cannot satisfy our expectations for wear leveling. Memory requests sizes are non-uniform distributed in the actual situation, which leads to a few free memory lists become hot lists. A few memory areas will still be worn out prematurely. To deal with this matter, UWLalloc limits the allocation times of each basic memory block. Each time a memory area is released, UWLalloc checks the wear count of each basic block. Blocks that reach to the wear limit will get isolated and become unallocable temporarily (see red block in Fig. 2). The remaining blocks are then merged with adjacent free and allocable blocks by checking the state list.

D. Major difference against existing wear leveling ware allocators

In literature, there are two wear leveling aware allocators, i.e., NVMalloc and Walloc. NVMalloc stipulates that a memory block should not be allocated twice within a time interval T, and its performance is highly sensitive to this parameter. Considering that the time patterns for different applications vary greatly from each other, it is difficult to determine an appropriate T in specific situations, as confirmed by experimental evaluation.

On the other hand, Walloc follows the less-allocated-first-out policy, and restricts the use of free lists whose average wear count exceeds a global threshold to keep them from being hot. However, some restricted blocks may become allocable accidentally when merged with the adjacent free blocks, resulting in an uneven wear inside some large size blocks, as confirmed by experimental evaluation.

Compared with NVMalloc, UWLalloc utilizes wear count instead of time interval to guide the memory allocation, which could achieve more stable wear leveling performance in various programs. And compared with Walloc, UWLalloc places a limit for each basic memory block, thus could provide more fine-grained wear leveling.

V. EXPERIMENT

A. Experiment setup

Two experiments are conducted to evaluate the wear leveling performance of our approach.

First, a random allocation test adopted in [10] [5] is utilized to help compare the wear leveling ability between *glibc malloc*, NVMalloc, Walloc and UWLalloc, which comprises 100K random memory allocation and deallocation operations (50% each). The allocation sizes are uniformly distributed between 10B and 1KB. And for each allocation, the entire allocated block is written once. The timestamp of NVMalloc is set to 10 microseconds while the global threshold for Walloc and wear limit for UWLalloc are both set to 100.

Next, applications of the above four allocators for the unified stack and heap managements are evaluated in term of wear leveling performance. Programs from Mibench [11] are used for the evaluation. A pin [12] based tool is developed to trace these programs' procedure calls, malloc requests and write operations. The uniform management process is then further simulated, where functions' frames are regarded as objects requesting for memory, like malloc requests; new memory space will be allocated and original write operations will be relocated. More allocations and deallocations will perform in this experiment, therefore we raise the timestamp of NVMalloc to 100 microseconds, the global threshold of Walloc and wear limit of UWLalloc to 200.

All experimental evaluations are performed on a linux virtual machine (kernel version 4.15.0, glibc version 2.23) with an Intel Core i5-7500 3.40 GHz CPU and 8 GB of DRAM memory. Since the PCM is not integrated along the DRAM in current commercial products yet, we use DRAM as a proxy.

B. Evaluation methodology

The target of wear leveling is to reduce the write variations and achieve an even usage of memory space. To quantify the variation, the *coefficient of variation* (CoV) is used as the evaluation metric, which is borrowed from [13], defined as follows:

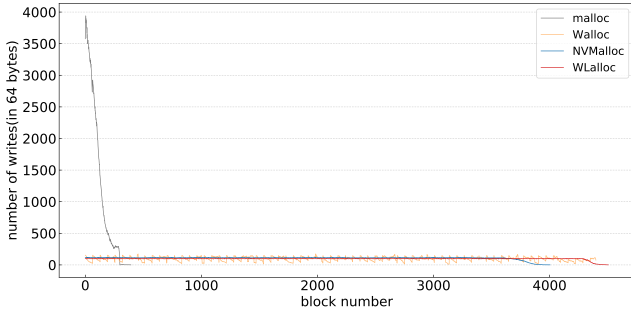
$$CoV = \frac{1}{w_{aver}} \cdot \sqrt{\frac{\sum_i^N (w_i - w_{aver})^2}{N-1}} \tag{1}$$

where w_i is the write count of the basic memory block (64 bytes) located at position i, w_{aver} is the average write count, and N is the total number of basic memory blocks.

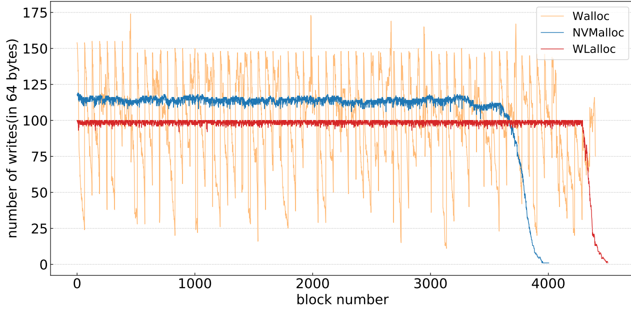
Based on the definition, a smaller CoV indicates the better wear leveling or a more even usage among memory cells. The value of CoV will be equal to zero when all the memory blocks are same in the write count.

C. Experimental result

a) Random allocation test: Fig. 3 shows the memory write patterns under random allocation of uniform distributed data size. From Fig. 3(a), we can see that NVMalloc, Walloc and UWLalloc write more evenly than *glibc malloc*, since malloc concentrates all the writes in a few memory areas. Fig.



(a) comparision between *glibc malloc*, NVMalloc, Walloc and UWLalloc



(b) close-up for NVMalloc, Walloc and UWLalloc

Fig. 3: Writes per block (64 bytes) under random allocation test

3(b) is a close-up for NVMalloc, Walloc and UWLalloc. Obviously, UWLalloc achieves the most uniform write distribution as the majority of memory blocks reaching or approaching the wear limit. Although the same adopting less allocated first out policy, Walloc ignores the uneven wear inside large memory blocks caused by accidentally reactivate restricted blocks, which leads to the squiggly memory usage. The CoVs for malloc, NVMalloc, Walloc and UWLalloc are 1.107, 0.217, 0.333 and 0.167 respectively, thus UWLalloc achieves a better wear leveling performance in this test case.

b) Uniform management for stack and heap: Fig. 4 shows the CoVs for each benchmark under the uniform stack and heap management with different allocators. Normalization is carried out in every benchmark. And the last column presents the average CoV for each allocator. The results show that, UWLalloc can obtain the smallest write variation in most cases, thus indicates a more even memory usage. Compared against *glibc malloc*, NVMalloc, and Walloc, UWLalloc can improve wear leveling by 41.9% (0.351 / 0.837), 30.3% (0.211 / 0.697) and 35.8% (0.271 / 0.757) on average.

It is found that *glibc malloc* performs the worst statistically, but it doesn't lag behind consistently. A typically example is *dijkstra*, where *glibc malloc* achieves the smallest CoV.

During the allocation, a wear leveling aware allocator attempts to allocates a memory block allocated less frequently, assuming that each block's write frequency is positively related to the allocation frequency. This assumption doesn't hold for *dijkstra*. Fig. 5 shows the allocation times of every 64 bytes block for different allocators in *dijkstra*. The distribution is seriously imbalanced in *glibc malloc*, while the others are

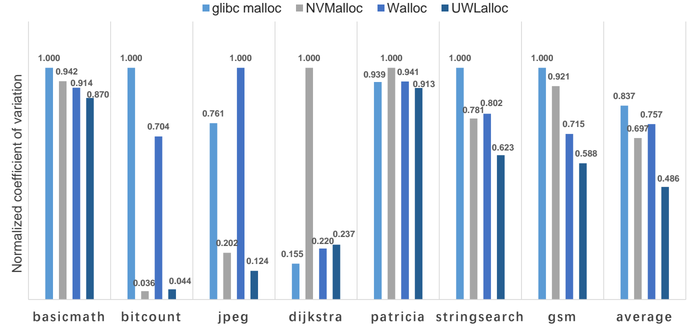


Fig. 4: CoVs under unified stack and heap management with different allocators

much more even. However, the non-uniformity of writes within the allocated memory blocks still cause an uneven usage. As shown in Fig. 6, some hot data within a function frame or a memory block imposes an intensive usage of a small fraction of memory cells, leading to the increment of CoV. Mitigating this kind of uneven writes requires wear leveling at a finer granularity than inter function frames or heap objects, which will be considered in future research.

Fig. 7 presents the memory usage under the uniform management for stack and heap. Malloc always leads to the least memory usage since it doesn't take wear leveling into consideration. Compared to NVMalloc and Walloc, UWLalloc allocates less memory space and at the same time achieves a better wear leveling performance.

VI. RELATED WORK

In the past few years, researchers proposed lots of wear leveling techniques to prolong the lifetime of PCM, from the perspective of hardware and software.

At hardware level, some hybrid main memory system architectures, consisting of DRAM and PCM, are designed in order to relegate most of write traffic from PCM to DRAM [14]. Row shifting, segment swapping [15], address remapping [16] and some other techniques are proposed to distribute write operations evenly among PCM cells. Although hardware

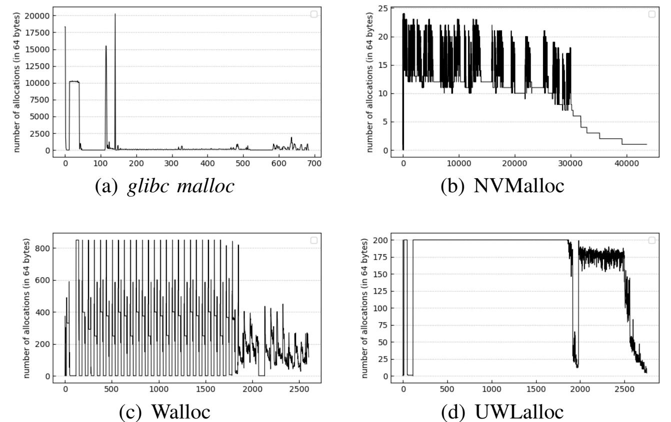
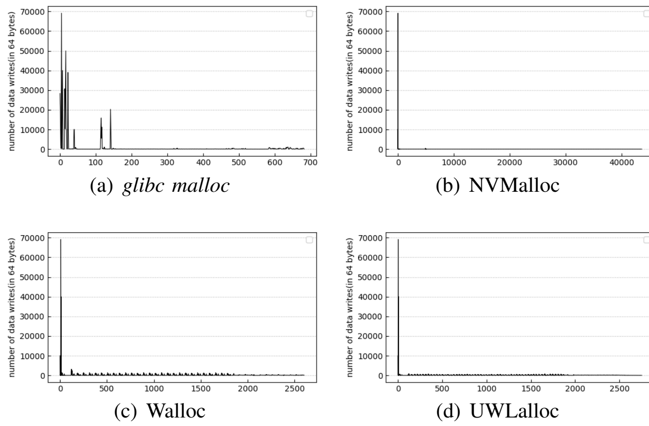


Fig. 5: Allocation times per 64 B block for *dijkstra*



(a) *glibc malloc* (b) NVMalloc

(c) Walloc (d) UWLalloc

Fig. 6: Writes per 64 B block for *dijkstra*

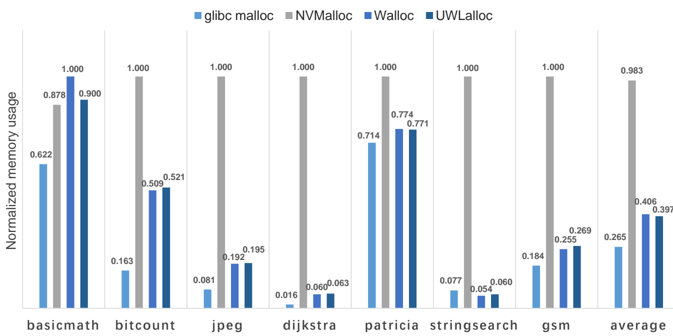


Fig. 7: Memory usage under unified stack and heap management with different allocators

wear leveling is necessary, it does not provide a complete solution. Some approaches shift data to new locations, which can impose overheads and reduce performance. Moreover, hardware techniques cannot well identify hot and cold data in programs, thus influencing the wear leveling performance.

On the other hand, software wear leveling complements hardware wear leveling for PCM [6] [7]. Some data allocation methods are proposed to help distribute memory writes evenly. Reference [9] proposed a Wear Leveling aware Dynamic Stack (WLDS) technique, which circularly allocates memory space to stack frames by next fit policy and mitigates the uneven wear in stack. Reference [10] [5] respectively proposed a wear aware memory allocator as the substitute for the conventional allocator, namely NVMalloc and Walloc, achieving an even usage of memory space in heap. However, these methods either consider stack or heap areas, but never both. And experiments show that, the uneven wear between stack and heap remains a pending problem.

VII. Conclusion

This paper proposes a new wear leveling aware allocator, and for the first time, this allocator provides a uniform management scheme for both stack and heap areas, thus could better balance writes in stack and heap areas. Experimental evaluations show that our allocator achieves a better performance than the state-of-the-art wear leveling aware allocators, and the uniform management of stack and heap obtains a significant reduction in write variation.

Acknowledgment

We thank anonymous reviewers for their insight and expertise comments that greatly improved this paper. This work is supported by the National Natural Science Found of China(No.61502346, 61640221, 61872272) and Hubei Provincial Natural Science Foundation of China(NO.2015CFB338).

References

[1] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Energy-efficient hybrid dram/nvm main memory," in *Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 492–493.

[2] K. T. Lim, P. Ranganathan, J. Chang, C. D. Patel, T. N. Mudge *et al.*, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *International Symposium on Computer Architecture (ISCA)*, 2008, pp. 315–326.

[3] R. Maddah, R. Melhem, and S. Cho, "Rdis: Tolerating many stuck-at faults in resistive memory," *IEEE Transactions on Computers*, pp. 847–861, 2015.

[4] M. Asadinia, M. Arjomand, and H. Sarbazi-Azad, "Variable resistance spectrum assignment in phase change memory systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 2657–2670, 2015.

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

[6] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: A hybrid pram and dram main memory system," in *Proceedings of the 46th Annual Design Automation Conference*. ACM, 2009, pp. 664–469.

[7] C. Pan, M. Xie, J. Hu, M. Qiu, and Q. Zhuge, "Wear-leveling for pcm main memory on embedded system via page management and process scheduling," in *Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2014, pp. 1–9.

[8] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, and E. H.-M. Sha, "Software enabled wear-leveling for hybrid pcm main memory on embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 599–602.

[9] Q. Li, Y. He, Y. Chen, C. J. Xue, N. Jiang, and C. Xu, "A wear-leveling-aware dynamic stack for pcm memory in embedded systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–4.

[10] 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 *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 2013, p. 1.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001, pp. 3–14.

[12] C.-K. Luk, R. Cohn, Muth *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*. ACM, 2005, pp. 190–200.

[13] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi, "i 2 wap: Improving non-volatile cache lifetime by reducing inter-and intra-set write variations," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 234–245.

[14] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Computer Architecture News*, pp. 24–33, 2009.

[15] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ACM SIGARCH computer architecture news*. ACM, 2009, pp. 14–23.

[16] G. Wu, H. Zhang, Y. Dong, and J. Hu, "Car: Securing pcm main memory system with cache address remapping," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. IEEE, 2012, pp. 628–635.