# A Wear-Leveling-Aware Dynamic Stack for PCM Memory in Embedded Systems

Qingan Li*†, Yanxiang He*, Yong Chen*, Chun Jason Xue†, Nan Jiang*, and Chao Xu*
*School of Computer Science, Wuhan University, China
†Department of Computer Science, City University of Hong Kong, Hong Kong
ww345ww@gmail.com, {yxhe,cyong}@whu.edu.cn, jasonxue@cityu.edu.hk, {nanjiang,xuch}@whu.edu.cn

***Abstract*—Phase Change Memory (PCM) is a promising DRAM replacement in embedded systems due to its attractive characteristics such as extremely low leakage power, high storage density and good scalability. However, PCM's low endurance constrains its practical applications. In this paper, we propose a wear leveling aware dynamic stack to extend PCM's lifetime when it is adopted in embedded systems as main memory. Through a dynamic stack, the memory space is circularly allocated to stack frames, and thus an even usage of PCM memory is achieved. The experimental results show that the proposed method can significantly reduce the write variation on PCM cells and enhance the lifetime of PCM memory.**

## I. Introduction

Due to increasing ubiquitous computing demands, Micro Controller Units (MCUs) have been widely adopted in everyday service devices, such as temperature sensors and MP3 players. MCUs usually integrate a very limited size of internal RAM on top of external Flash storage, which exposes Flash to heavy write traffic. Recent works have proposed to deploy emerging Phase Change Memory (PCM) as an universal memory to replace internal RAM and external Flash [1] [2] [3]. Compared to DRAM, PCM has better scalability, comparable read latency and lower leakage power consumption. Compared to Flash, PCM enjoys faster read/write access speed and much longer cell endurance. Previous works [4] [5] [6] have shown that a PCM main memory can achieve significant energy saving with comparable performance to that of a DRAM main memory.

However, PCM cells suffer from limited lifetime. Furthermore, the non-uniformity in writes to different PCM cells can marvellously reduce the achievable lifetime. A system built on pure PCM based memory may fail after several days because of the broken PCM cells. To address this issue, researchers have proposed lots of methods which can be classified into two classes. The first class is to reduce the total number of bit-level writes. Differential writes methods are proposed in [7] [8] [9]. Upon each write request, these methods compare the new value with the old value, and re-write only the different bits. The second class is wear leveling [5] [10]. Wear leveling techniques distribute write operations evenly among PCM cells to avoid the phenomenon that a small proportion of PCM cells become too hot to wear out at an early stage. All these works require customized hardware design.

Many MCUs are configured without Memory Management Unit (MMU) to achieve better trade-offs among cost, performance and energy efficiency, such as Cortex-M0 and Cortex-M1 [11]. For these MCUs, a software enabled wear leveling method is proposed by Hu et al. [12]. Their method targets to reduce the write variation in static storage space. The basic idea of this method is to re-allocate the data objects at the entry and exit of each code region (a function or a loop). This method employs an array to record the number of writes for each memory address, and re-allocates an object to a memory address if the total write count in this address is not over a threshold value. However, this method works well only for situations when the memory addresses can be known at compilation time. It is not applicable for stack space. This is because, the address of a function's frame is only determined at run time. The uneven usage of stack space leads to uneven usage of PCM cells. We conducted a set of experimental evaluation of the usage of PCM cells, and observed that the uneven usage of stack space is mainly due to the stack allocation mechanism itself. Based on this observation, this paper proposes a wear leveling aware dynamic stack for PCM memory in embedded systems to achieve an even usage of stack space.

The reminder of this paper is organized as follows. Section II discusses the background as well as the motivation. Section III introduces the Wear Leveling aware Dynamic Stack (WLDS) method. Section IV discusses the experimental evaluation. Section V concludes this paper.

## II. Background and Motivation

In this section, we first present a brief introduction to the conventional stack memory allocation, and discuss the reasons for the uneven usage of stack space. Then, we present a brief discussion on dynamic memory allocation. Finally, a motivating example is presented to show the potential optimization via dynamic stack allocation.

### A. Stack memory allocation

Each function instance is associated with a frame (also called *active record*) to store the context information for this function. Local data, including local variables and compilation temporary variables, are stored in this frame. A conventional stack based allocator works as follows:

This work was partially supported by the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 123210, CityU 123811], and the National Natural Science Found of China [Project No. 61170022, 61373039, 91118003].

978-3-9815370-2-4/DATE14/©2014 EDAA

1) A specific memory address is assigned to the *main* function's frame, which can be viewed as the start point of the stack.
2) Upon a function is called, the callee function's frame is allocated on top of the caller function's frame.
3) Upon a function returns, the callee function's frame is deallocated from the top of caller function's frame.

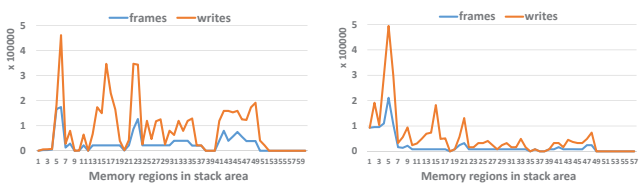
The composition of the allocated frames constitutes the stack area. Since frame allocation/deallocation is always conducted on the top of stack, some memory regions may be used by a large number of frames, while others are rarely used. Furthermore, due to the conservative reservation of stack space, a proportion of memory space, sometimes a sizeable proportion, at the low end of the reserved space are never used. Therefore, the stack allocation mechanism itself contributes much to the uneven usage of stack space.

Fig. 1 shows the memory usage with the conventional stack allocation. It shows the number of writes to each memory region within the stack area, as well as the number of frames mapped to each memory region. There are two observations. First, the stack area is used extremely unevenly, which is harmful to PCM memory's lifetime. Second, if a memory region receives lots of writes, the writes often come from lots of frames. In other words, since multiple frames are mapped to a single region, the write count is large. If these frames are mapped more evenly to regions, the memory usage can be much more even. Based on this observation, this paper proposes a Wear Leveling aware Dynamic Stack (WLDS) method to evenly use the memory space.

### B. Dynamic memory allocation

The proposed WLDS method relies on dynamic memory allocation to evenly map frames to memory regions. Here we present a brief discussion on dynamic memory allocation. A typical dynamic allocator works as follows:

1) A free block list is maintained to record the unused memory blocks.
2) Upon a memory allocation request, the allocator does a search of the free list and chooses a block that fits. There are several selection policies. The *first fit* policy searches the free list from the beginning and chooses the first free block that fits. The *next fit* policy is similar to *first fit*, but starts search where the previous search stops. The *best fit* policy examines every free block and chooses the one with the smallest size that fits. Since this policy requires an exhaustive search of the list, it is usually slow.




(a) (b)

Fig. 1. Uneven memory usage in stack area. It shows the number of writes as well as frames over stack area. (a). Memory usage of *basicmath*. (b). Memory usage of *FFT*.

3) Upon a request of deallocation of a block, the allocator inserts the block back to the free list and optionally coalesces it with its adjacent blocks.

The *best fit* policy is rarely used in practice due to its high cost. On the contrary, the *first fit* policy is widely used in many allocators, such as the buddy system allocator. However, the *first fit* policy leads to a higher use of the blocks at the beginning of the free list and a lower use of the blocks at the end of the free list. As a result, the *first fit* policy is not suitable for wear leveling. In contrast, the *next fit* policy searches the free block list circularly to find free blocks that fits, which can provide an even memory usage. This paper employs the *next fit* policy to design the wear leveling aware dynamic stack.

### C. A motivating example

As stated above, the uneven usage of stack space is mainly due to the stack allocation itself. This issue can be addressed by replacing this stack allocator with a dynamic stack allocator which uses the memory space in a circular fashion.

Here a motivating example is presented to show the potential optimization space. A sample code is illustrated in Fig. 2, where the *main* function invokes function *f*, and then invokes function *g*, which further invokes function *h* three times. Assume that all function frames are of the same size. Fig. 3(a) shows the situation when the conventional stack allocator is applied. The frame of the *main* function is allocated first. Then the frames of function *f* and *g* share the same memory space. Note that the frames of *f* and *g* can share the same memory address, since *f* and *g* have disjoint life ranges. Finally, three frames of function *h* share the same memory addresses. The shared addresses are prone to be write intensive since they are written by more function instances.

Alternatively, each function frame can use disjoint memory space, as illustrated in Fig. 3(b). Using a dynamic stack allocator, each memory region is allocated to at most one function frame, and thus the wear leveling is improved. However, this allocator assumes a large enough stack space.

As a comparison, Fig. 3(c) shows a dynamic allocator which provides a better trade-off between wear leveling and stack space requirement. Assuming that the design reserves a stack space of four frames. This dynamic allocator can circularly allocate the free stack space to function frames. With this allocator, each memory region is allocated to at most two function frames, with a controllable expansion of memory footprint.

```
1  void main()
2  {
3      f();
4      g();
5  }
6  void g()
7  {
8      int i = 0;
9      for(; i < 3; i ++)
10         h();
11 }
```

Fig. 2. A sample code.

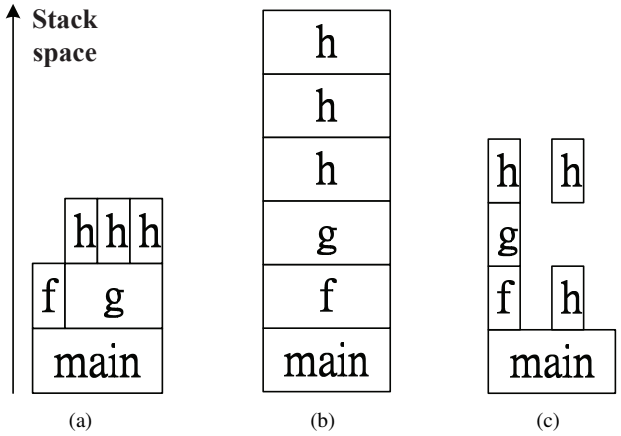



Fig. 3. Comparison of stack allocators. (a). The conventional stack allocator. (b). The dynamic stack allocator assuming unlimited stack space. (c) The dynamic stack allocator with stack space limitation.

## III. Wear Leveling aware Dynamic Stack

As stated above, it is common that the stack space is unevenly used due to the stack allocation itself as well as the various application features. This section proposes a Wear Leveling aware Dynamic Stack (WLDS) method to evenly use the stack space. This method employs a customized dynamic stack allocator to achieve a uniform mapping from function frames to stack space. Together with a re-allocation technique, which maps a hot local data to different addresses in different code regions to achieve an even usage within a function frame, this method can obtain a significant reduction in write variation. In this section, we first discuss the dynamic stack allocator in detail. Then we discuss the application of this allocator to stack area. Finally, we present a brief discussion on applying the re-allocation technique to achieve an even usage within a function frame.

### A. Dynamic stack allocator

The desired dynamic stack allocator works as follows. Immediately before a function *f* executes, there is a memory request for *f*'s frame. Upon this request, the allocator searches the free block list for *f*'s frame using a *next fit* policy. The *next fit* policy is employed to use the stack space circularly. Immediately after a function *f* finishes its execution, its frame is deallocated and the corresponding memory block is released back to the free block list. A coalescing operation is optionally conducted to coalesce continuous free blocks.

In other words, the dynamic stack allocator implements two functions, *alloc* and *dealloc*. The *alloc(int nSize)* function allocates stack space to a frame with the size of *nSize* via a *next fit* search of the free list. Upon each memory request, it starts the search of the free block list from the position where the previous search stops. Before checking whether a block fits, it first recursively coalesces this block with the succeeding neighbours. The *dealloc(int nAddr)* frees the memory block with the address of *nAddr* and inserts the corresponding block back into the free block list in order of addresses.

### B. Application to stack area

To apply the dynamic stack allocator to function frames, the function invocation needs to be wrapped with additional operations. An extra entry, control link (CL), should be added to the frame to store the frame address of the caller function. The wrapping works as follows. Assume that there is a caller function $f$ invoking a callee function $g$, and the (highest) memory address for the caller function $f$'s frame is $\alpha$.

1) Immediately before the invocation of $g$, the *alloc* function is employed to obtain the highest memory address of $g$'s frame, $\beta$.
2) The value of $\alpha$ is stored in $g$'s frame, which is used later for searching function $g$'s frame. To support the storage of the frame address of the caller function, $\alpha$, an extra entry, CL, is added into the frame.
3) The *ebp*'s content is updated with $\beta$. Here *ebp* represents the register holding the stack base pointer. With this operation, $g$'s frame is layout at address $\beta$.
4) Function $g$ executes afterwards.
5) Immediately after $g$ returns, before any other operations, *ebp*'s content is updated with the CL value, $\alpha$,to point back to $f$'s frame.
6) Finally, the *dealloc* function is employed to free the block used by $g$'s frame.

### C. Even memory usage within a function frame

With the dynamic stack technique proposed above, the memory space can be circularly used by function frames during run time. As a result, a uniform mapping from function frames to memory space can be achieved. However, the non-uniformity of writes within the local stack area may still cause an uneven usage of stack space. For example, a hot local data in a function frame imposes an intensive usage of a small fraction of memory cells. To deal with this issue, we apply the method proposed in [12] to re-allocate hot local data into multiple addresses among the frame for different code regions.

## IV. Experiment

This section discusses the experimental evaluation. First, the experimental methodology and setup are introduced. Then, the experimental results are presented.

### A. Experimental methodology

The objective of wear leveling is to reduce memory write variation and uniformly distribute write traffic. To quantify the memory write variation, we define the *coefficient of variation* (CoV) based on standard deviation. The CoV is defined as follows:

$$CoV = \frac{1}{E} \cdot \sqrt{\frac{\sum_{i}^{N} (w_i - E)^2}{N-1}} \quad (1)$$

where $w_i$ is the write count of a memory line labelled $i$, $E$ is the average write count defined using Equation 2, and $N$ is the total number of memory lines. As it is defined, a smaller CoV indicates a better wear leveling or a smaller write variation among memory cells. An ideal wear leveling technique makes the write count the same among all memory

lines, and thus CoV is zero. This definition of CoV is employed in the following experiments to evaluate the proposed wear leveling method.

$$E = \frac{1}{N} \cdot \sum_{i}^{N} w_i \tag{2}$$

### B. Experimental setup

This paper evaluated three methods.

- The first method (SA) is the baseline which employs the conventional stack allocation.
- The second method (RA) is similar to the SA method, but it employs the software technique proposed in [12] to re-allocate data objects for each code region in purpose of evenly using the space within a frame.
- The third one is the proposed Wear Leveling aware dynamic stack method (WLDS), which employs a customized dynamic stack allocator to circularly map frames to stack space, together with the re-allocation technique to evenly use the space intra a frame.

We have developed a pin [13] based tool to implement all three methods mentioned above. Applications from MiBench [14] are used for the experimental evaluation.

### C. Experimental results

Fig. 4 shows the CoVs for each benchmark under stack space limitation of 512 KB. Note that a smaller CoV indicates better wear levelling or a more even memory usage. The results show that, the conventional SA method always leads to the largest write variation. The RA method, which re-allocates local data within a function frame can significantly reduce the write variation. The proposed WLDS method, which achieves a uniform mapping from function frames to memory space via a customized dynamic stack and achieves a uniform local stack area usage via re-allocating local data, can obtain the smallest write variation in most cases.

It is a little strange that for *bitcount* and *susan*, the RA method works a little better than the WLDS method. The reason is that, for these benchmarks, there are extremely hot frames. In the RA method, a memory region is allocated to the hot frames alone. But, in the WLDS method, to achieve a uniform mapping, this memory region is also allocated to a small number of other frames, which makes this region hotter.

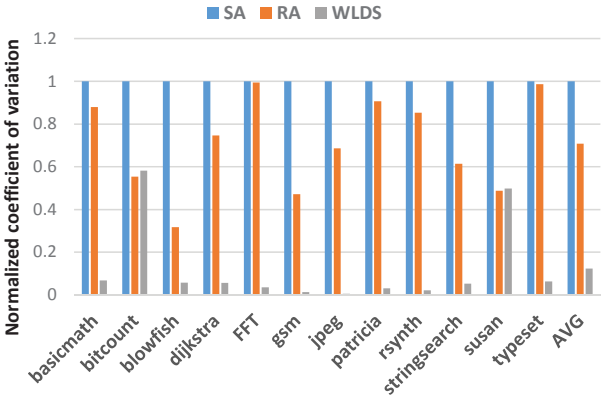



Fig. 4. CoVs under stack space limitation of 512 KB. All results are normalized to the SA method.

## V. Conclusion

This paper proposes a pure software wear leveling method to reduce the write variation on PCM memory in embedded systems without MMU. This method employs a customized dynamic stack to achieve a uniform mapping from function frames to memory space. This technique, together with the intra-frame re-allocation technique, obtains a significant reduction in write variation.

## References


[1] T. Liu, Y. Zhao, C. J. Xue, and M. Li, "Power-aware variable partitioning for dsps with hybrid pram and dram main memory," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. ACM, 2011, pp. 405–410.

[2] J. Hu, C. Xue, Q. Zhuge, W.-C. Tseng, and E.-M. Sha, "Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, 2011, pp. 1–6.

[3] P. Zhou, Y. Zhang, and J. Yang, "The design of sustainable wireless sensor network node using solar energy and phase change memory," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, 2013, pp. 869–872.

[4] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: a hybrid pram and dram main memory system," in *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*. IEEE, 2009, pp. 664–669.

[5] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 2–13, 2009.

[6] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li, "Emerging non-volatile memories: Opportunities and challenges," in *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '11. ACM, 2011, pp. 325–334.

[7] 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*, vol. 37, no. 3. ACM, 2009, pp. 14–23.

[8] W. Zhang and T. Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 2–13.

[9] S. Cho and H. Lee, "Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 347–357.

[10] 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 *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 14–23.

[11] "Cortex-m specification summary," ARM Holdings. [Online]. Available: http://arm.com/products/processors/cortex-m

[12] 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 *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. IEEE, 2013, pp. 599–602.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[14] 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.