2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

AIMR: An Adaptive Page Management Policy for Hybrid Memory Architecture with NVM and DRAM

Zhiwen Sun Zhiping Jia Xiaojun Cai Zhiyong Zhang Lei Ju*
School of Computer Science and Technology
Shandong University, China
E-mail: julei@sdu.edu.cn

***Abstract*—The last few years have witnessed the emergence of Non-Volatile Memories (NVMs), which are actively pursued as scalable substitutes for traditional DRAM-based main memory due to higher scalability and lower leakage power. However, current NVM technologies also exhibit potential drawbacks including lower endurance, higher dynamic power, and longer write latency. Recent studies show that hybrid memory architectures involving NVM and DRAM are able to effectively utilize the merits of both memory devices. However, allocating write-intensive pages to NVM can still greatly impeding the performance of the overall memory system. In this paper, we provide an adaptive page management policy called AIMR (Adaptive page Insertion, Migration, and Replacement) for hybrid memory architecture. The main objective of our scheme is to ensure DRAM absorbs most memory writes while maintaining a high performance of the overall system. Specifically, we use both "recency" and "frequency" features to estimate the future memory access patterns and then carefully insert, migrate, and replace pages in the hybrid memory without setting any additional user-defined parameters. Experimental results show that AIMR can achieve average NVM write count, memory access latency, and memory energy consumption reduction by 19.3%, 45.8% and 27.8%, respectively compared with the existing page management policies under hybrid memory architecture.**

I. Introduction

Embedded systems need denser, lower power, and more reliable main memory devices due to the tight power and scale budgets. Existing DRAM memory not only consumes up to 40% of system energy [1]–[3], but also suffers from limited scalability below 16 nm resulted from the constraints of leakage current and capacitor placement. In addition, the refresh cycle time increases with the capacity of DRAM device [4]. Fortunately, emerging non-volatile memory (NVM) technologies that show better scalability, lower static power, and comparable performance as DRAM, such as Phase Change Memory (PCM) [5] and Resistive RAM (ReRAM) [6] bring a new perspective of resolving memory-related issue in future embedded system. However, there are two major drawbacks for current NVM technologies that hinder the adoption of a pure-NVM based main memory. First, a write access to NVM incurs more dynamic energy and latency. Second, the endurance of NVM is limited compared with DRAM.

In order to take the beneficial characteristics from both NVM and DRAM, the hybrid architectures have been proposed in [7]–[10]. Generally, pages with more read operations are placed in NVM, while write-intensive pages are placed in DRAM so as to mitigate their impact on NVM's performance and endurance. In general, two types of hybrid architecture have been presented. The first architecture depicted in Fig. 1(a) uses a small DRAM as the buffer of NVM [11]–[13]. Since the buffer is managed as an inclusive hardware cache, the DRAM space does not add to the overall memory capacity. More importantly, for workloads with poor locality, the cache actually lowers performance and increases energy consumption. The second architecture depicted in Fig. 1(b) puts the NVM and DRAM on the same level of memory hierarchy [7], [8]. In this architecture, both NVM and DRAM constitute the overall memory address space and migrations are introduced to make frequently written pages in the DRAM, leaving read-intensive pages in the NVM. However, those migrations can induce much excrescent CPU control and memory accesses.

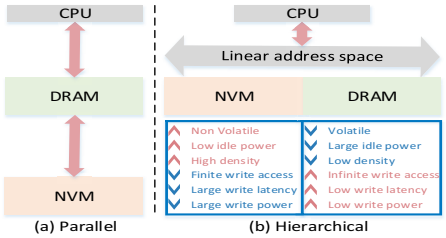

Fig. 1. Comparison of architectures involving both NVM and DRAM. (a) Parallel Organization; (b) Hierarchical Organization.

In this paper, we focus on the hybrid memory architecture depicted in Fig. 1(b). Generally, two problems exit in this architecture. First, more migrations may be caused if a page load into a improper memory device from the auxiliary storage. Second, inaccurate prediction of write-intensive pages may cause many bad migrations. Traditional page management policies for DRAM-only memory are not suit to this architecture because they are designed for memory of uniform access latency, dynamic power, and unlimited endurance. On the other hand, existing page management policies designed specifically for hybrid memory architecture have three common defects [10], [14], [15]. First, when a page inserted into the hybrid memory, those policies have very little information about the future access pattern of that page. This may lead to an improper insertion decision on whether the page should be loaded into NVM or DRAM. Second, always caching read-intensive pages in NVM may lead to additional writes to NVM especially in read-intensive workloads. Finally, these policies need some pre-defined parameters and no single a priori choice works uniformly well across various workloads and memory sizes. In this paper, we provide a novel page management policy called AIMR (Adaptive page Insertion, Migration, and Replacement) for hybrid memory architecture. Compared to the previous work, our contributions are summarized as follows.

*Corresponding author

978-1-4799-8937-9/15 $31.00 © 2015 IEEE
DOI 10.1109/HPCC-CSS-ICESS.2015.179

LRU-WPAM：分成四个队列。NVM和DRAM中各两个；
每个page都有一个权值，随着请求的类型，权值会增加；
当超过某一设定阈值时，会发生迁移page。
缺陷：需要提前设定参数，无法自适应。

(1) **A page insertion algorithm**. When a page inserted into memory from auxiliary storage, we explore the access pattern of that page by using the history lists and insert the page into DRAM if it is write-intensive in the past.

(2) **A page migration algorithm**. We use both "recency" and "frequency" features to identify write-intensive pages. We only migrate write-intensive NVM pages to DRAM without migration of read-intensive pages from DRAM to NVM. Thus we can make the use of the DARM space more efficiently.

(3) **A page replacement algorithm**. When the memory is full, we replace either "recency" or "frequency" referenced pages according to the memory access pattern of the workload.

We use gem5 [17] integrated with NVMain [18] to run SPEC2006 benchmark suit [20] in order to test the performance of AIMR. We conduct comparisons with other policies including CLOCK, CLOCK-DWF [14], and LRU-WPAM [15]. Experimental results show that AIMR achieves average NVM write count, memory access latency, and memory energy consumption reduction by 19.3%, 45.8% and 27.8%, respectively. Besides, AIMR is self-tuning without any need for workload specific priori knowledge.

自适应

II. Related Work

Recently, several page management policies have been proposed for NVM/DRAM hybrid memory architecture.

RaPP [10] features a hardware-driven page placement policy. The policy relies on the memory controller (MC) to monitor access patterns, migrate pages between DRAM and NVM, and translate the memory addresses coming from the cores. Periodically, the operating system updates its page mappings based on the translation information used by the MC.

CLOCK-DWF [14] manages memory pages in NVM and DRAM separately. That paper analyzes the characteristics of memory write references and find that write frequency is generally a better estimator than temporal locality in predicting the re-reference likelihood of write references. In CLOCK-DWF, when a page fault occurs, if the request is read, the page is put on NVM; otherwise the page is put on DRAM. When a page on NVM hits by write request, the page is migrate to DRAM. To get a free page frame while the memory is full, NVM uses conventional CLOCK algorithm but DRAM migrate a low write frequency page to NVM. Consequently, CLOCK-DWF basically has the following weaknesses. First, improper placement of a page when it is first load into memory will cause many additional migrations. Second, it may introduce a lot of unnecessary migrations since it always causes migrations if write to a NVM page. Third, CLOCK-DWF force read-bound pages be placed on NVM, when most operations of the workload are read, those pages cannot be placed on DRAM. This may incur higher NVM writes than conventional policies.

LRU-WPAM [15] is based on the LRU replacement algorithm but add prediction and migration schemes. LRU-WPAM aligns all pages in the hybrid memory as a LRU queue, and use four monitoring queues, DRAM read queue, DRAM write queue, NVM read queue and NVM write queue. Each page is retained into both LRU list and one of the four queues according to its access pattern and memory type. The algorithm provides a weight value for each page. Each time a page hits in the memory, the pages weight is calculated again according to the type of this access request, if its weight value exceeds the threshold, the page will be migrated. If the memory need to choose a victim to release for receiving the migrated page, DRAM chooses the least recently used page in DRAM read queue and NVM choose the least recently used page in NVM write queue. LRU-WPAM needs to determine many off-line parameters (α, Tr_{mig}, and Tr_q). However, no single choice works uniformly well across different workloads and memory sizes.

根据访问模式和内存类型，每个页面都保留在LRU列表和四个队列之一中。

Next, we describe our page management policy, called AIMR (Adaptive page Insertion, Migration, and Replacement), which manages pages without the limitations of previous works. AIMR adopts the superiorities of CLOCK-DWF and LRU-WPAM but removes the defects mentioned above. Besides, AIMR have constant-time implementation complexity and low space overhead.

优势

III. AIMR

Basically AIMR contains three algorithms. An algorithm is for *Page Insertion* to select the appropriate memory device when a page load into hybrid memory from auxiliary storage. An algorithm for *Page Migration* to identify each write-intensive page in NVM and select a page in DRAM to achieve intermigration. An algorithm for *Page Replacement* to evict a page when the memory is full.

Now we explain the intuition behind AIMR. Let S denotes the total hybrid memory size, then our policy AIRM manages $2S$ pages. S pages in memory managed by T1 and T2. S metadata of recently evicted pages managed by B1 and B2. Pages evicted from T1 are placed on B1, and those evicted from T2 are placed on B2. For each page in T1 or T2, we will maintain a *reference bit* and a *dirty bit*. Besides, for each page in T2 or B2, we will maintain a *suggest bit*. All of those bits can be set to either one or zero. A page in T1 is promoted to T2 only it can be referenced frequently and recently. We think T1 captures "recency" while T2 captures "frequency". History lists (B1 or B2) only keep the metadata about the recent evicted pages. History information are used for two purpose:

(1) To guide a continual adaptive process that keeps readjusting the sizes of T1 and T2 like ARC [16]. AIMR maintains a target size *TargetSize* for T1. By implication, the target size for T2 will be S - *TargetSize*. If a reference hits on B1, then we increase *TargetSize*, otherwise if hits on B2, then we decrease *TargetSize*.

B1/B2历史表的作用：

(2) To record the historical write characteristics of recently evicted pages. According to CLOCK-DWF [14], write frequency is generally a better estimator than temporal locality in predicting the re-written likelihood. So we keep a *suggest bit* in T2 and B2 since they capture the "frequency" feature. If a page access hits B2 and the *suggest bit* of that page is one, then we should insert it from auxiliary storage into DRAM.

在预测重写可能性时，写频率通常比时间局部性更好的估计量！！！

For a visual description of AIMR, See Fig. 2. We adopt the following invariants and definitions.

$$0 \le |T1| + |T2| \le S \;\; ; \;\; 0 \le |T1| + |B1| \le S \tag{1}$$

$$0 \le |T1| + |T2| + |B1| + |B2| \le 2S \tag{2}$$

$$|T1| + |T2| < S \Leftrightarrow B1 \cup B2 = \emptyset. \tag{3}$$

$$|T1| + |B1| + |T2| + |B2| \ge S \Leftrightarrow |T1| + |T2| = S. \tag{4}$$

$$0 \le T1_n + T2_n \le S_{nvm} \;\; ; \;\; 0 \le T1_d + T2_d \le S_{dram} \tag{5}$$

$$T1_d + T1_n = |T1| \;\; ; \;\; T2_d + T2_n = |T2| \tag{6}$$

自适应的判断page的写入位置；
根据历史表B1/B2判断。

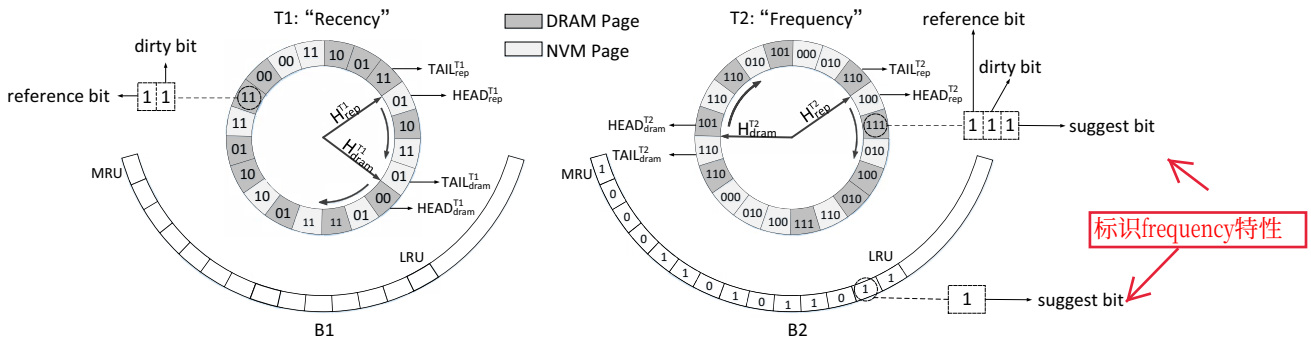

Fig. 2. A visual description of AIMR. The CLOCK lists T1 and T2 contain pages that are in hybrid memory and LRU lists B1 and B2 contain recent evicted pages from T1 and T2, respectively. T1 represents pages of “recency” feature while T2 represents pages of “frequency” feature. A page in T1 is promoted to T2 only when it is within the *frequency filter window*. The sizes of the T1 and T2 are adapted continuously in response to the evolving workload. When a hit in B1 is observed, the target size of T1 is incremented; similarly, when a hit in B2 is observed, the target size of T1 is decremented. When there comes a page fault, the insert algorithm inserts the missed page into the $TAIL_{rep}^{T1}$ if the missed page hits B1 or totally new (neither in B1 nor B2) while inserts into the $TAIL_{rep}^{T2}$ if the missed page hits B2. When migrating a write-intensive NVM page inside Tx (x is 1 or 2) to DRAM, H_{dram}^{Tx} rotates clockwise. If it encounters a DRAM page with dirty bit is 1, then reset it and keeps rotating, otherwise the migrate algorithm has get a DRAM page to make the intermigration with the write-intensive NVM page. When the hybrid memory is full, a page in T1 should be evicted to B1 if the length of T1 is above the target size, otherwise, a page in T2 should be evicted to B2. If H_{rep} encounters a page whose reference bit is 1, then reset it to 0 and keeps rotating, otherwise the replace algorithm has get a victim page and evict it to Bx.

S_{nvm} and S_{dram} are the total NVM and DRAM size, respectively. $T1_n$ and $T2_n$ are the size of NVM pages in T1 and T2, respectively. $T1_d$ and $T2_d$ are the size of DRAM pages in T1 and T2, respectively.

经常写入的page、（11X）in T2

Definition 1. (Write Hot Page). *Write Hot Page is defined as the set of T2 pages whose reference bit is one and dirty bit is one. A page can become a Write Hot Page only this page has been written frequently and recently.*

Definition 2. (Bad Migration). *A Bad Migration occurs in one of two cases: (1) when a page originally in NVM is migrated to DRAM and then back to NVM without being written enough times while in DRAM; (2) when a page originally in DRAM is migrated to NVM and then back to DRAM with too many write accesses while in NVM.*

Before the introduction of our policy, we elaborate on the two windows used in this paper.

frequency filter window. To avoid the side-effect of *correlated references*, we set this window to the time span for H_{rep} turning one round in T1. This is equivalent to access a page with *reference bit* is 1. 1x

write-intensive filter window. To enable early migrations while trying to avoid *Bad Migration*, we set this window to the time span for H_{dram} turning one round in T2. This is equivalent to a write access occurred on *Write Hot Page*. 11x

We now examine our policy AIMR in detail based on the pseudo-code in Algorithm 1.

Initially, we set the *TargetSize* to 0 and lists T1, B1, T2, B2 to empty set. When there comes a page access x and the operation is *op*, we check whether x hits T1 (line 1). If so, we check if x is within the *frequency filter window* (line 2-3). If hits T2 (line 8), we migrate NVM pages within the *write-intensive filter window* to DRAM inside T2 (line 11-13). Besides, we set the *suggest bit* to 1 if x is a DRAM page which in the *write-intensive filter window* (line 11, 12, 14). Note that we don’t migrate when a page hits on T1. This is because if the page is write-bound in T1, it will be linked to T2 soon. This can greatly cut down the *Bad Migrations* and greatly reduce the migrate number.

If happens a page miss (line 19), we first check if there is any free page (we first check for free DRAM page since DRAM has high static power but low active power). However, if the hybrid memory is full, line 23 carries out the *replace* procedure to get a free page. The replace algorithm may cause the history (B1 or B2) full and if so, we replace the history by the rule similar to ARC [16]. That is: If |T1| + |B1|

Algorithm 1 AIMR (page x, operation *op*)

Initialization: Set *TargetSize* = 0; T1, B1, T2, B2 = $\emptyset$;
1: **if** $x \in$ T1 **then** /* page hit */
2: **if** *reference_bit* is 1 **then**
3: Reset *reference_bit*, *dirty_bit* and link it to $TAIL_{rep}^{T2}$;
4: **else**
5: Set *dirty_bit* to 1 if *op* is write;
6: Set *reference_bit* to 1;
7: **end if**
8: **else if** $x \in$ T2 **then** /* page hit */
9: **if** *op* is read **then**
10: Set *reference_bit* to 1;
11: **else** /* op is write */
12: **if** x is a *Write Hot Page* **then**
13: if $x \in$ NVM, then *page_migrate(p, T2)*;
14: if $x \in$ DRAM, then set the *suggest_bit* to 1;
15: **else**
16: Set x to a *Write Hot Page*;
17: **end if**
18: **end if**
19: **else** /* page miss */
20: **if** there is a free page **then**
21: page pointer p point to a free page;
22: **else** /* memory is full, evict a page */
23: page pointer p = *page_replace()*;
24: **end if**
25: Discard a history page if history is full;
26: *page_inserte (x, op, p)*;
27: **end if**

contains exactly S pages, then remove the LRU page from B1, otherwise, if |T1| + |B1| + |T2| + |B2| contains exactly $2S$ pages, then remove the LRU page from B2 (line 25). Finally, we insert the page x into the free page p (line 26).

A. Page Insertion Algorithm

If there comes a cache miss and we get a free page p from hybrid memory. But we should not insert the requested page x into p immediately to avoid placing write-intensive pages into NVM. In AIMR, if one of the following conditions happens, we should insert x into DRAM.

插入DRAM的两个条件：

Cond.1 The page x is in B2 and the *suggest bit* of x is one.

Cond.2 The page x is neither in B1 nor B2 and *op* is write.

However, if p points to a NVM page and one of above two conditions happens, then migrate algorithm in AIMR gets a DRAM page q and insert the x into q. Specifically, migration is carried out inside T2 for *Cond.1* and inside T1 for *Cond.2*. We now examine the insertion algorithm in Algorithm 2. Lines

Algorithm 2 page_insert (page x, operation *op*, free page p)

```
1: if x ∉ B1 ∪ B2 then  /* x is totally new */
2:     if op is write and p ⊂ NVM then  /* Cond.2 */
3:         p = page_migrate(p,T1);
4:     end if
5:     Insert x to p and make p the TAIL_rep^T1;
6: else if x ∈ B1 then  /* x hits B1 */
7:     TargetSize = min{TargetSize + max{1, |B2|/|B1|}, S};
8:     Insert x to p and make p the TAIL_rep^T1;
9: else  /* x hits B2 */
10:    TargetSize = max{TargetSize - max{1, |B1|/|B2|}, 0};
11:    if suggest_bit of x is 1 and p ⊂ NVM then  /* Cond.2 */
12:        p = page_migrate(p, T2);
13:    end if
14:    Insert x to p and make p the TAIL_rep^T2;
15: end if
16: Set reference_bit, dirty_bit and suggest_bit of x to 0;
```

1-5 illustrate the procedure in the case that the requested page is totally new. We make it the $TAIL_{rep}^{T1}$(line 5). Line 6 checks x hits B1, then we increase the *TargetSize* for T1 (line 7) and make it the $TAIL_{rep}^{T1}$(line 8). The role of our adaptation is to "invest" in the list that is most likely to give the highest hit per additional page invested. Similarly, if x hits B2 (line 9), then we decrease the *TargetSize* for T1 (line 10) and make it the $TAIL_{rep}^{T2}$(line 14). In the end, we set the flag bits to 0 (line 16). Note that even *op* is write, we still set the *dirty bit* to zero when insertion because a page inserted into memory from auxiliary storage always trigger a memory (NVM or DRAM) write. Thus we can postpone the migrate procedure and greatly cut down *Bad Migration*.

B. Page Migration Algorithm

页面迁移：只从NVM迁移到DRAM；不会从DRAM迁移到NVM

Different from the existing policies designed for hybrid architecture, the migration algorithm in AIMR only migrate write-bound pages from NVM to DRAM but doesn't try to put read-bound pages from DRAM to NVM. Thus, AIMR can make full use of the DRAM space even for read-intensive workloads. If one of the following conditions happen, then AIMR triggers the migration procedure.

Cond.1 When a page fault occurs and we get a free NVM page p. If the missed page x is totally new (not in B1 or B2) and the operation type *op* is write, according to the temporal locality (recency), the migrate algorithm should choose a DRAM page q in T1 and migrate the data in q to p. Finally, we insert x into q.

Cond.2 When a page fault occurs and we get a free NVM page p. The missed page x is B2 and the *suggest bit* is one. Then the migrate algorithm chooses a DRAM page q in T2 and migrate the data in q to p. Finally, we insert x into q.

Cond.3 When a NVM page p is within the *write-intensive filter window* in T2, we think it is a frequently written page. Then, the migrate algorithm chooses a DRAM page q in T2 and make the intermigration with p.

Algorithm 3 page_migrate (page pointer p, list Tx)

```
1: while (true) do  /* the migration procedure inside Tx */
2:     q = H_dram^Tx and H_dram^Tx points to the next DRAM page;
3:     Reset the dirty bit of all the NVM pages that H_dram^Tx bypassed
       to 0.
4:     if dirty_bit of q is 0 then
5:         In Cond.1, migrate q to p and link p to TAIL_dram^T1;
6:         In Cond.2, migrate q to p and link p to TAIL_dram^T2;
7:         In Cond.3, make a intermigration between q to p;
8:         Return q;
9:     else
10:        Set dirty_bit of q to 0;
11:    end if
12: end while
```

H_{Tx} dram该指针指向的是T1中的DRAM页，且是顺时针旋转。

Algorithm 3 illustrates the migrate algorithm inside the list Tx (x is 1 or 2). We use H_{dram}^{Tx} to keep looping until find a DRAM page for migration (line 1-11). This procedure is similar to the traditional CLOCK algorithm but page pointer H_{dram}^{Tx} only points to DRAM pages and we use *dirty bit* to provide second chance. We carry out the migration procedure inside a specific list (T1 or T2), but if there is no DRAM page in this list, the migration will be abolished. However, we have haven't seen this scenario (场景) in our experiments.

C. Page Replacement Algorithm

The replace algorithm is simple but adaptive. That is, if the actual length of T1 exceeds (超过) *TargetSize*, then replace a page in T1. otherwise, replace a page in T2. Algorithm 4 shows the details of replacement procedure. Line 2-8 describes the procedure of replacing a page in T1. The procedure of replacing a page in T2 is similar.

Algorithm 4 page_replace ()

```
1: while True do
2:     if |T1| ≥ max(1,TargetSize) then
3:         p = H_rep^T1 and H_rep^T1 points to the next page;
4:         if the reference_bit of p is 0 then
5:             Reclaim p and make it the MRU page in B1;
6:         else
7:             Reset the reference_bit of p;
8:         end if
9:     else
10:        evict a page in T2 and make it the MRU page B2;
11:    end if
12: end while
```

回收页面p；并且将p插入B1的MRU端。

IV. Experiments

In this section, we present the performance evaluation results of different page management policies.

缺陷：DRAM容量相比NVM要小，需要执行从DRAM到NVM的迁移，来释放DRAM的空间。

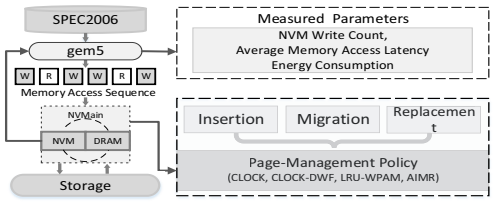

Fig. 3. Simulation for evaluation of page management policies.

A. Simulation Setup

We modify gem5 [17] integrated with NVMain [18] as our simulator platform. Fig. 3 shows the overall simulation environment. The size of each page is set to 4 KB, which is common to most operating systems. The baseline configurations are listed in Table I. Both DRAM and NVM in this paper are modeled similarly to [19]. Generally, we set the size of NVM four times as large as DRAM.

TABLE I. System Settings

Processor (ARM)	6 cores, 2 GHz, out-of-order, issue width=8
L1 I&D-cache	Private, 1KB per core, 2-way, 64-byte lines
L2 cache	Shared, 2MB, 16-way, 64-byte lines
Memory Configuration	512MB, 2 ranks/DIMM, 8 chips/rank, 8 banks/chip
DRAM Delay&Timing (cy)	tRCD: 5, tCL: 5, tWL: 4, tCCD: 4, tWTR: 3, tWR: 6, tRTP: 3, tRP: 5, tRRDact: 3, tRRDpre: 3
DRAM Energy (pJ/bit)	Array read: 1.17, Array write: 0.39, Buffer read: 0.93, Buffer write: 1.02, Background power: 0.08
NVM Delay&Timing (cy)	tRCD: 22, tCL: 5, tWL: 4, tCCD: 4, tWTR: 3, tWR: 6, tRTP: 3, tRP: 60, tRRDact: 2, tRRDpre: 11
NVM Energy (pJ/bit)	Array read: 2.47, Array write: 16.82, Buffer read: 0.93, Buffer write: 1.02, Background power: 0.08
Storage drive	5 ms average access latency

Table II summarizes the SPEC2006 CPU benchmarks we used in this paper. The six benchmarks are run in parallel with reference input size. we use 500 million instructions for the cache warmup following 6 billion instructions for the statistics (each benchmark runs 1 billion instructions).

TABLE II. Benchmarks Used in This Paper

Benchmarks	unique pages	read reference	write reference
mcf	49974	70724693	13162003
bwaves	18726	12748672	2913304
libquantum	18454	203181203	78490640
soplex	95258	125880882	47445263
sjeng	49097	749599200	712342647
lbm	154728	284217060	206135953

B. Write Count on NVM

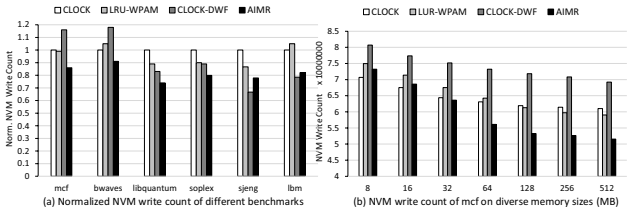

Fig. 4. Write count on NVM.

Write count on NVM is important because it is related to the life time of NVM. Fig. 4(a) shows the NVM write count normalized to CLOCK for each benchmark under hybrid memory. We can see AIMR causes the minimum NVM writes for most benchmarks (mcf, bwaves, libquantum, soplex). This is because CLOCK-DWF cannot make use of the DRAM space in those read-intensive benchmarks for it only insert the write pages into DRAM, but most accesses are read and this may trigger many page replacements in NVM. Besides, too many migrations also greatly increase the NVM writes. The same situation also happens to LRU-WPAM. On average, AIMR can reduce the total NVM write count by 19.3% compared to CLOCK-DWF while 15.5% compared to LRU-WPAM.

We also measured with diverse memory sizes for benchmark *mcf* shown in Fig. 4(b), we choose *mcf* for it can better highlight the advantages of our policy compared to the other benchmarks. For all policies, NVM write count decreases as the memory size grows. This is because the DRAM size is increased accordingly. Besides, when the memory is small, All of LRU-WPAM, CLOCK-DWF and AIMR induce higher NVM writes than CLOCK for they all trigger many migrations which add many excrescent NVM writes.

C. Migration Count

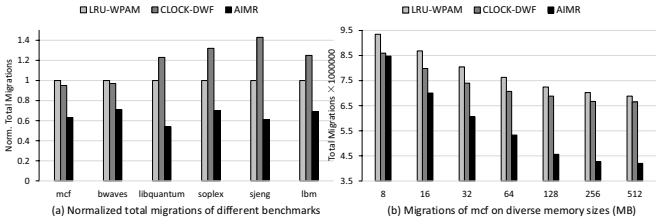

Fig. 5. Migrations between NVM and DRAM.

Fig. 5(a) shows the total migrations normalized to LRU-WPAM for each benchmark under hybrid memory. We can see AIMR conducts the smallest migrations for all benchmarks. This is because for each write to NVM, CLOCK-DWF will trigger a migration and LRU-WPAM not only migrates write-intensive pages from NVM to DARM but also migrates read-intensive pages from DRAM to NVM.

Fig. 5(b) shows the migrations with diverse memory sizes for benchmark *mcf*. We can see with the increasing of hybrid memory size, migration number decreases because the DRAM size increased.

D. Average Memory Access Latency

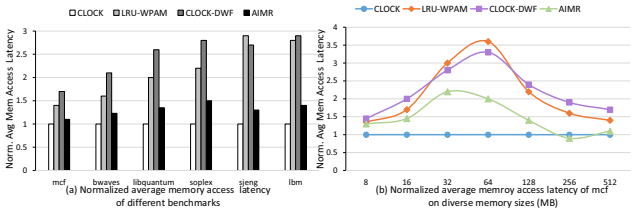

Fig. 6. Average Memory Access Latency.

Fig. 6(a) shows the average memory access latency of LRU-WPAM, CLOCK-DWF, and AIMR under hybrid memory normalized to CLOCK under DARM-only memory. We can see AIMR causes the smallest average memory access latency than CLOCK-DWF and LRU-WPAM for all benchmarks. This is because AIMR conducts fewer page faults and migrations than them. On average, AIMR can reduce the average memory access latency by 45.8% compared to CLOCK-DWF and 35.7% compared to LRU-WPAM.

page fault:缺页异常

Fig. 6(b) shows the normalized average memory access latency with diverse memory sizes for benchmark *mcf*. When the memory size is extremely small, the latency of all policies are almost identical because of too many page faults. The latency disperse when the memory size increases. This is

because all of LRU-WPAM, CLOCK-DWF, and AIMR trigger many migrations. There is an interesting phenomenon that when the memory size is 256MB, AIMR causes smaller latency than CLOCK for AIMR has few page faults and has not too many migrations since the memory size is large.

E. Total Memory Energy Consumption

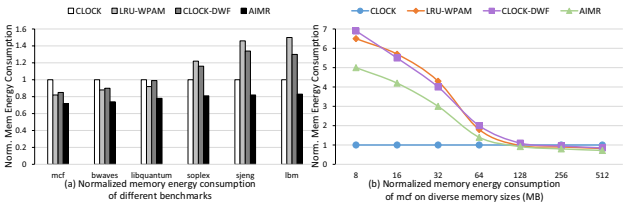

Fig. 7. Total consumed energy of AIMR, LRU-WPAM, and CLOCK-DWF on hybrid memory normalized to CLOCK on DRAM-only memory.

Let us move to the energy consumption, which consists of active energy and static energy. The active energy refers to the energy which is consumed when the data moves either into or out of the memory devices. The static energy is due to the transistor leakage, peripheral circuitry, the data refresh operations, and the total elapsed time of the benchmark.

Fig. 7(a) shows the energy consumption of LRU-WPAM, CLOCK-DWF, and AIMR under hybrid memory normalized to CLOCK under DARM-only memory. We can see AIMR consumes the lowest energy compared to others. This is because AIMR conducts almost the smallest NVM writes, migrations and total elapsed time than the others. In total, AIMR can reduce the total energy consumption by 24.6% on average and up to 47.4% compared to CLOCK-DWF while 27.8% on average and up to 39.6% compared to LRU-WPAM.

Fig. 7(b) shows the normalized energy consumption with diverse memory sizes for benchmark *mcf*. When the memory size is extremely small, the power consumption of the hybrid architecture is larger than that of the DRAM-only memory architecture. This is because the active power of NVM is larger than DRAM and the static energy consumption is low since the total memory is small. However, when memory size becomes large, static energy consumption required for DRAM refresh operations accounts for a large portion of energy consumption compared to the active power consumption required for actual read/write operations.

V. Conclusion

This paper proposed an adaptive page management policy for NVM/DRAM hybrid memory to combine the advantages of both when it is adopted in embedded system. It can balance "frequency" and "recency" features to dynamically evolving workloads. AIMR doesn't need any off-line selection for crucial tunable parameters and works well across workloads with diverse memory access pattern and locality. Besides, AIMR has const implement complexity and low space overhead. AIMR achieves average NVM write count, memory access latency, and memory energy consumption reduction by 19.3%, 45.8% and 27.8%, respectively compared to the existing policies LRU-WPAM and CLOCK-DWF. Finally, Considering the characteristics of flash storage [21], [22] to improve the performance of AIRM would be an interesting direction for us to explore.

VI. Acknowledgements

This research is sponsored by the Natural Science Foundation of China (NSFC) under Grant No. 61202015, the National 863 Program 2015AA011504, Shandong Provincial Natural Science Foundation under Grant No. ZR2013FM028, and the Fundamental Research Funds of Shandong University under No. 2015JC030.

References

[1] J. Stuecheli, D. Kaseridis, H. C. Hunter, and L. K. John, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 375–384.

[2] A. N. Udipi, N. Muralimanohar *et al.*, "Rethinking dram design and organization for energy-constrained multi-cores," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 175–186, 2010.

[3] T. Zhang, M. Poremba *et al.*, "Cream: a concurrent-refresh-aware dram memory architecture," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 368–379.

[4] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," in *IEEE International Symposium on Computer Architecture (ISCA)*, 2012, pp. 1–12.

[5] O. Zilberberg, S. Weiss, and S. Toledo, "Phase-change memory: An architectural perspective," *ACM Computing Surveys (CSUR)*, vol. 45, no. 3, p. 29, 2013.

[6] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 476–488.

[7] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 2, p. 6, 2011.

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

[9] 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*, vol. 37, no. 3, pp. 24–33, 2009.

[10] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *ACM Proceedings of the international conference on Supercomputing*, 2011, pp. 85–95.

[11] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated control for energy-efficient and heterogeneous memory systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA2013)*, 2013, pp. 424–435.

[12] H. G. Lee, S. Baek, C. Nicopoulos, and J. Kim, "An energy-and performance-aware dram cache architecture for hybrid dram/pcm main memory systems," in *IEEE International Conference on Computer Design (ICCD)*, 2011, pp. 381–387.

[13] 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, 2009, pp. 14–23.

[14] S. Lee, H. Bahn, and S. Noh, "Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2187 – 2200, 2013.

[15] H. Seok, Y. Park, K.-W. Park, and K. H. Park, "Efficient page caching algorithm with prediction and migration for a hybrid main memory," *ACM SIGAPP Applied Computing Review*, vol. 11, no. 4, pp. 38–48, 2011.

[16] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies, FAST'03*, vol. 3, 2003, pp. 115–130.

[17] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[18] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2012, pp. 392–397.

[19] 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.

[20] J. L. Henning, "Performance Counters and Development of SPEC CPU2006," *ACM SIGARCH Computer Architecture News,* vol. 35, no. 1, pp. 118-121, 2007.

[21] R. S. Chen, Z. Qin, Y. Wang, D. Liu, Z. Shao, Y. Guan, "On-Demand Block-Level Address Mapping in Large-Scale NAND Flash Storage Systems," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1729 – 1741, 2015.

[22] R. S. Chen, Y. Wang, J. Hu, D. Liu, Z. Shao, Y. Guan, "Unified non-volatile memory and NAND flash memory architecture in smartphones," in *Asia and South Pacific Design Automation Conference, ASP-DAC'20*, 2015, pp.340-345.