NVM/DRAM Hybrid Memory Management with Language Runtime Support via MRW Queue

Gaku Nakagawa
Department of Computer Science
University of Tsukuba
JSPS Research Fellow
gnakagaw@cs.tsukuba.ac.jp

Shuichi Oikawa
Division of Information Engineering
Faculty of Engineering, Information and Systems
University of Tsukuba

***Abstract*—Non-volatile memory (NVM), such as PCM, STT-MRAM, and ReRAM, makes it possible to integrate secondary storage into main memory. This integration reduces I/O access times to typically slow block devices; however, it is unrealistic to construct a large capacity main memory with a single NVM at this time, because NVM have disadvantages regarding write access. Combining NVM and other memory devices is necessary to hide such disadvantages. In particular, we should place write-hot data on DRAM and write-cold data on NVM. For data placement, programming language runtime supports are useful, since they have more detailed information about write access than the operating system. A previous study proposed a method to manage NVM/DRAM hybrid memory with programming language runtime supports, determining data placement based on the number of write accesses to each object (i.e., the individual counting method); however, this approach has two problems, namely memory efficiency and determination of threshold values for data placements. The Most Recent Write (MRW) queue method is an alternative method to distinguish between write-hot data and write-cold data. MRW queue manages the frequency of write accesses to objects. In this study, we discuss the problems of the individual counting method and show solutions using the MRW queue approach. Results of our experimentation show that the MRW queue method improves memory efficiency and reduces overhead of the individual counting method.**

I. Introduction

Several research projects focused on new generation non-volatile memory (NVM), including STT-MRAM [1], PCM [2], and ReRAM [3]. These NVM can be used as main memory, because they are byte accessible devices. Non-volatile main memory makes it possible to integrate secondary storage into main memory. Currently, main memory comprises DRAM, which is a volatile memory device. Therefore, persistent data must be stored in secondary storage devices, which include hard disk drives (HDDs). It will be possible to record persistent data in main memory with NVM. Such integration reduces I/O access times to slow block devices.

While it is possible to record persistent data in main memory with NVM, it is not possible to construct a large capacity main memory with a single NVM at this point. Instead, we must combine DRAM and NVM or combine NVM and another NVM to construct non-volatile main memory. In general, we must consider three key requirements when choosing a memory device for main memory. First, the device should have high I/O performance. Second, the device should have a large capacity. Third, the device should have high write endurance. Currently, NVM do not fulfill any of these three requirements.

Combining multiple memory devices makes it possible to meet the above requirements. There are several proposed methods in the several research work on NVM/DRAM hybrid main memory architectures, which combine PCM and DRAM [4]–[8]. PCM has limitations regarding write access, i.e., large write latencies and low write endurance. To overcome these limitations, the allocator places write-hot data on DRAM and write-cold data on NVM. Note that write-hot data is data with many write accesses, whereas write-cold data is data with few write accesses. This data placement policy hides the limitations of NVM.

Previous studies advocated that programming language runtime supports are useful for managing NVM/DRAM hybrid main memory [9]. Memory allocation for NVM/DRAM hybrid memory is based on the characteristics (i.e., frequency) of write accesses. Several research endeavors [4]–[8] have proposed to collect write access characteristics and determine the placement of data at the operating system (OS) level. Language runtimes, such as the Java Virtual Machine (JVM), have more detailed information regarding write access to data than hardware or the OS. Language runtime supports are useful for managing NVM/DRAM hybrid memory. In [9], Nakagawa and Oikawa proposed a method to manage NVM/DRAM hybrid main memory based on the number of write accesses to each object, i.e., the individual counting method. Using this method, language runtimes determine the placements of data between NVM and DRAM. Such placements are based on write access characteristics and the semantics of the data. Results of their evaluation experiments showed that the individual counting method reduced write accesses to the NVM [9].

The individual counting method [9] has two problems. First, the method worsens memory efficiency. Recording the number of write accesses requires additional memory space. The individual counting method adds a write access counter area to the header of each object, thus increasing the size of all objects. Furthermore, the language runtimes do not use all the added area, because there are objects without any write access; therefore, such memory is wasted.

Second, it is difficult to determine the threshold of data placement dynamically. Programming language runtimes execute various programs, and each program has its own characteristics of write access to data. The language runtimes should

978-1-4799-8676-7/15/$31.00 copyright 2015 IEEE
SNPD 2015, June 1-3 2015, Takamatsu, Japan

dynamically determine the threshold for data allocation; however, dynamic determination is difficult when only counting individual write accesses.

To resolve these problems, we designed an alternative method to distinguish write-hot and write-cold data. The language runtimes do not record the number of write accesses to each object in our new method. Language runtimes manage write-hot data with a specific queue data structure called the Most Recent Write (MRW) queue. Using the MRW queue, write-hot data are placed at the top of the queue. In contrast, write-cold data are placed at the bottom of the queue. Our new method does not require individual write access counting for each object. Furthermore, language runtimes dynamically determine the threshold for migration.

In this paper, we describe the individual counting method [9] and the problems associated with it, as well as our MRW queue method. We performed an experiment to evaluate our MRW queue method. Results show that our new method improved memory efficiency by 2.9%-14.3%. Furthermore, our new method reduced overhead by 0.74%-10.17%.

The remainder of this paper is structured as follows. In Section 2, we describe the background of this study, including language runtime support for NVM/DRAM hybrid memory, the individual counting method, and the problems associated with it. In Section 3, we detail our MRW queue method. In Section 4, we describe our experiment and summarize results. Based on our experimental results, we discuss the impact of our MRW queue method in Section 5. In Section 6, we describe related work. And in Section 7, we conclude our paper and suggest future work.

II. BACKGROUND

A. Language Runtime Support for NVM/DRAM Hybrid Memory

Operating systems manage the placement of data on NVM/DRAM hybrid memory, as described in [4]–[8]. OS collects write access characteristics of data and determines the placement of data based on these characteristics. In this subsection, we describe how language runtime support is useful for managing NVM/DRAM hybrid memory.

Language runtime support is useful for managing NVM/DRAM hybrid main memory for two reasons. First, language runtimes can more efficiently place data. In OS-level methods, the OS manages the placement of data in a fixed unit, such as a page unit. Furthermore, the OS collects characteristics of write access in this fixed unit. If a page contains only a few write-hot data, memory allocation may be inefficient. For example, if we assume that a page contains one write-hot data and 10 write-cold data, it is preferable that the 10 write-cold data are placed in NVM space; however, the OS places the page in DRAM, because the OS concludes that the entire page comprises write-hot data. As a result, the write-cold data are placed in DRAM. In contrast, language runtimes manage memory allocation in flexible units. They can also collect characteristics of write accesses in such flexible units. The language runtimes make it possible to efficiently place data in situations that write-hot data and write-cold data are mixed.

The second reason that language runtime support is useful is that data can be placed on the basis of their semantics. Data have various semantics. It is possible to predict the characteristics of write access based on the semantics of data. In particular, the semantics of data are an important information for determining the placement between NVM and DRAM. At the OS level, it is difficult to distinguish the semantics of data. The OS cannot typically determine the placement of data based on the semantics of data. In contrast, language runtimes can distinguish types of each data. Therefore, language runtime support is useful for managing NVM/DRAM hybrid main memory architectures.

B. The Individual Counting Method

In [9], Nakagawa and Oikawa proposed the individual counting method for the efficient placement of data on NVM/DRAM hybrid memory with language runtime support. The language runtimes determine the placement of data based on the number of write accesses to each object. In this subsection, we describe the individual counting method.

The individual counting method utilizes object-oriented programming language runtime information associated with generational garbage collection. Garbage collection is the automatic reclamation of heap-allocated storage after its last use by a program [10]. Generational garbage collection is one of many garbage collection methods. Generational garbage collection makes use of trends in object lifetimes. In particular, most objects become garbage in a short time period, while long-lived objects continue to live.

In generational garbage collection, objects are placed in two distinct heaps, namely the young area and the old area. All objects are placed in the young area when they generated, then once the language runtime detects an object as being valid for a while, the language runtime regards the object as a long-lived object. Such long-lived objects are moved to the old area, a process called promotion. The young area and the old area are managed with appropriate garbage collection methods. In general, the young area is managed via a copying garbage collection scheme, whereas the old area is managed using mark-sweep garbage collection. Furthermore, garbage collection of the young area is called minor garbage collection, while garbage collection of the old area is called major garbage collection.

The individual counting method comprises the following four processes: (1) collecting the characteristics of write accesses; (2) determining data placements in object generation; (3) determining data placements in object promotion; and (4) determining object migration after object promotion.

1) Collection of the write access: The individual counting method determines data placement based on the characteristics of write accesses. The language runtimes use a write barrier mechanism to collect such characteristics for generational garbage collection. The write barrier mechanism detects write accesses to objects. The individual counting method utilizes this write barrier mechanism to collect the characteristics of write accesses.

Each object has an associated counter that records write counts. The language runtimes increment the counter when

a write access is detected. The write characteristics of data semantics are also important to collect. The language runtimes collect write access characteristics of class type, as well as the characteristics of individual objects.

2) Data placements in object generation: All newly generated objects are placed on DRAM. The language runtimes place all new objects in the young area, which is a write-hot area; in particular, language runtimes write to the young area with high frequency. The language runtimes generate and dispose huge amounts of objects in the young area. The language runtimes also perform many write accesses to achieve garbage collection in this area. The young area is unsuitable for placement on the NVM. Therefore, the language runtimes place all new data on DRAM.

3) Data placements in object promotion: The language runtimes determine the placement of objects in object promotion by moving long-lived objects from the young area to the old area (i.e., object promotion). The language runtimes determine the placement of objects based on the class type of the target objects. When the target object is in the write-hot class, the language runtimes place the data on DRAM. In contrast, when the target object is in the write-cold class, the language runtimes place the data on the NVM.

4) Object migration after promotion: The language runtimes migrate objects between NVM and DRAM after placement and object promotion, as necessary. The language runtimes may have write-hot objects on the NVM or write-cold objects on DRAM. It is preferable that write-hot objects are on DRAM and that write-cold objects are on the NVM. Therefore, the language runtimes should migrate objects, as necessary.

The language runtimes migrate write-hot objects on NVM to DRAM. To do so, the language runtimes first detect target objects via the write barrier mechanism. When the language runtimes detect a write access to an object, it examines the write access counter of the target object. If the target object is a write-hot object on the NVM, the language runtimes mark the target object as a migration target. Note that the language runtimes do not migrate objects when target objects are detected. Instead, target objects are simply marked. The language runtimes migrate marked objects when a major garbage collection is executed. The purpose of this delayed migration is to reduce the overhead involved with updating references. The process of object migration comprises not only copying data, but also updating references to target objects. The language runtimes must scan all objects to find any and all references to the target objects. This examination can cause huge overhead.

The language runtimes move the marked objects and update the references to the target objects during major garbage collection. The language runtimes scan all objects to identify valid objects when major garbage collection is executed. The individual counting method utilizes this scanning. During a scan of all objects, the language runtimes can migrate objects marked multiple times.

The language runtimes migrate write-cold objects on DRAM to the NVM. In such cases, the language runtimes cannot detect target objects via the write barrier mechanism. The language runtimes detect the write-cold objects on DRAM via major garbage collection by examining the write count of objects on DRAM during the major garbage collection scan. When the language runtimes detect a write-cold object on DRAM, the language runtimes migrate the object to the NVM.

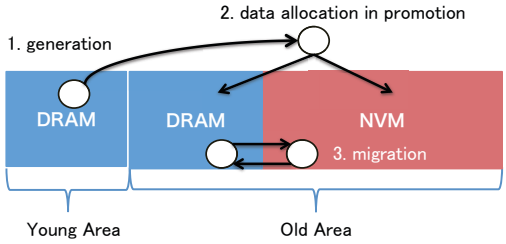

Fig. 1. Overview of The Individual Counting Method

Figure 1 illustrates the process flow of the individual counting method. As mentioned above, all objects are generated in the young area. The language runtimes place all young objects on DRAM, which corresponds to step 1 of the figure. Long-lived objects are moved to the old area, and the language runtimes place target objects on the NVM or DRAM, which corresponds to step 2 of the figure. The language runtimes determine the placement of objects based on the class types of the target objects. Further, the language runtimes migrate objects between the NVM and DRAM, as necessary, which corresponds to step 3 of the figure.

C. Problems of the Individual Counting Method

The individual counting method has two fundamental problems. First, the method worsens memory efficiency. Since this method requires all objects to have write access counters, extra memory space is required for each object header to record the corresponding write access counter. Given that the number of objects is huge, the extra space consumed comprise a substantial amount of memory. Furthermore, such memory usage is useless for write-cold objects.

Second, it is difficult to dynamically determine the threshold for object migration. In the individual counting method, the language runtimes migrate write-hot objects on the NVM to DRAM. The language runtimes should dynamically determine the threshold for this migration, because the language runtimes execute various programs, each with its own write access characteristics; however, it is not realistic to determine the threshold based on the individual write access count. In the individual counting method, each object has information regarding the write access. To determine the threshold, a scan of all objects is required. This process is costly. After the information is collected via this scan, it must then be analyzed. This process also is costly. As an example, sorting object references based on their write counts consumes huge resources. For these reasons, the individual counting method uses a static threshold for the migration of write-hot objects.

III. Most Recent Write (MRW) Queue Method

Our MRW queue method can resolve the problems note above regarding the individual counting method. Using our method, the language runtimes manage information regarding write-hot objects with a special queue data structure. The language runtimes don't record individual write access counts,

thus reducing the extra memory required to record write counts for each object. The MRW queue also refers to the data structure used to dynamically record write-hot objects. The language runtimes detect write-hot objects without a costly object scan. Thus, the MRW queue method resolves the problems of the individual counting method.

A. Most Recent Write Queue

The MRW queue is a special queue data structure that manages the frequency of write accesses to objects. Figure 2 illustrates the structure of the MRW queue. From the figure, the MRW queue has n entries to record references to objects; n is the MRW depth parameter. Each entry has an entry number. Assuming there is entry q, entry a is the upper entry ($a < q$) and entry b is the lower entry ($q < b$).

When the language runtimes detect a write access to an object, the MRW queue is scanned. If an entry for the target object is found, it swaps that entry for the neighboring upper entry, as shown in Figure 3. If an entry for the target object is not found, a new entry is inserted into the MRW queue, as shown in Figure 4. The new entry indicates the target object, with the position of the insert noted as the MRW insert point parameter. When the language runtimes insert an entry, all lower entries are shifted down, with the entry at the bottom being disposed.

B. Utilization of MRW Queue

The MRW queue method uses this MRW queue to manage the frequency of write accesses to objects. In the individual counting method, when the language runtimes detect a write access, they update the corresponding write counter associated with that target object. In contrast, using our MRW queue method, only the MRW queue is updated. After a while, references to write-hot objects concentrate near the top of the MRW queue. References to write-cold objects concentrate near the bottom of the MRW queue. Note that the language runtimes only record information regarding objects on the NVM, not caring about objects on DRAM.

The language runtimes detect write-hot objects on the NVM based on the MRW queue. When the language runtimes execute a major garbage collection, the MRW queue is scanned. The language runtimes regard entries from 0 to n as write-hot objects and mark them as targets of migration, as depicted in Figure 5. This n value is deemed the MRW threshold parameter. The language runtimes set the migration marks of the target objects, which is the same process as in the individual counting method. The language runtimes migrate the marked objects from the NVM to DRAM during a major garbage collection.

The language runtimes migrate the write-cold objects on DRAM to the NVM. In the individual counting method, the language runtimes detect write-cold objects based on the write counters of objects on DRAM. In our MRW queue method, the language runtimes cannot use the write counter information. The objects in our method have a 1-bit Boolean flag that describes whether the object has been written to at least once. The language runtimes use this information to detect write-cold objects on DRAM. The language runtimes execute this detection process during a major garbage collection. When a target object is detected, it is migrated from DRAM to the NVM.

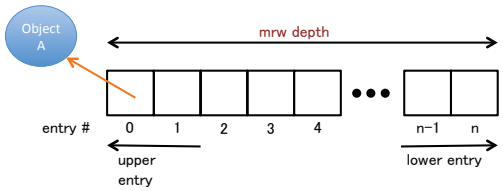

Fig. 2. Most Recent Write queue (MRW queue)

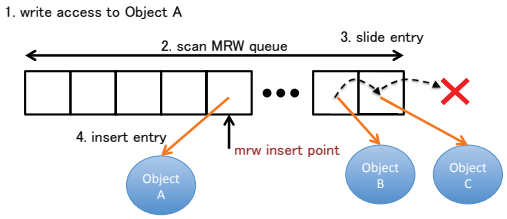

Fig. 3. update MRW queue (swap)

IV. Experiment

We implemented our MRW queue method on the Jikes Research Virtual Machine (Jikes RVM) [11]. The Jikes RVM is a Java language runtime. The base version of Jikes was 10709 (mercurial commit number). We performed an experiment to evaluate the impact of our MRW queue method as compared to the individual counting method. Therefore, we had two implementations, one being the individual counting method, the other our MRW method. We executed a benchmark set on these two implementations and recorded the following information: the number of write accesses to DRAM or the NVM; execution times; heap memory usage; and so on. The benchmark set was the dacapo benchmark suite [12], version 2006-10-MR2. This benchmark suite has 11 benchmark programs. We used all programs except for the chart benchmark programs, because chart benchmarks did not work in our experimental environment. In our experiments, we did not use any NVM devices. All data were placed on normal DRAM. We divided the heap into two separate areas, one regarded as DRAM space, the other as NVM space. Furthermore, we ignored other characteristics of NVM devices, such as write access latency and so on.

Each implementation had a variety of parameters. For the individual counting method, there were two parameters, namely NVM_MIGRATION_THRESHOLD and FORCE_MAJORGC_THRESHOLD. The NVM_MIGRATION_THRESHOLD parameter represents the threshold value for determining the objects to move from the NVM to DRAM. The FORCE_MAJORGC_THRESHOLD parameter represents the threshold value for forcing a major garbage collection to occur when the program has many write accesses to the NVM. In our experiments, NVM_MIGRATION_THRESHOLD was set to 100, and FORCE_MAJORGC_THRESHOLD was set to 100,000.

For our MRW queue method, there were four parameters, namely MRW_DEPTH, MRW_THRESHOLD, MRW_

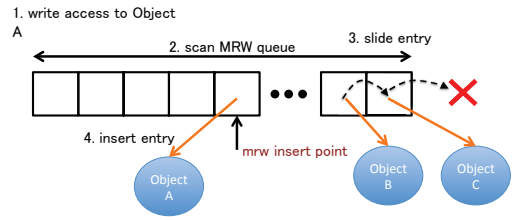

Fig. 4. update MRW queue (insert)

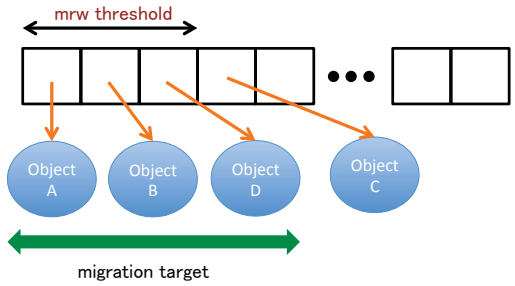

Fig. 5. determine migration target

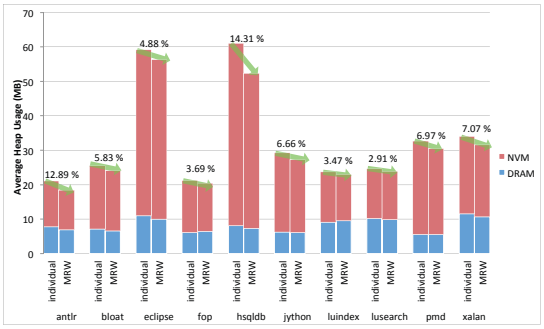

Fig. 6. Average Heap Size

TABLE I. MAXIMUM HEAP SIZE

benchmark	individual (MB)	MRW (MB)	reduction (%)
antlr	28.58	23.40	18.12
bloat	34.34	31.07	9.52
eclipse	71.57	75.85	-5.98
fop	29.62	28.22	4.73
hsqldb	100.62	85.91	14.62
jython	31.93	29.94	6.55
luindex	31.44	33.16	-5.47
lusearch	30.06	29.38	2.26
pmd	37.20	36.64	1.51
xalan	40.14	38.80	3.34

INSERT_POINT, and FORCE_MARJORGC_THRESHOLD. In our experiments, MRW_DEPTH was set to 100, MRW_THRESHOLD was set to 40, MRW_INSERT_POINT was set to 100, and FORCE_MAJORGC_THRESHOLD was set to 100,000.

We evaluated the following four indexes: (1) the average size of the heap area; (2) the maximum size of the heap area; (3) the execution time; and (4) the number of write accesses to memory. The implemented language runtimes change heap size during the execution of the benchmark programs. We measured the heap sizes when the language runtimes performed a minor garbage collection, then averaged them. The maximum size of the heap area was the largest heap size of the measured values. The execution time is the time to complete each benchmark program. In our experiments, we added several processes to record statistical data, including heap size and so on. These processes are unnecessary for our proposed methods and were added merely for evaluation purposes. Furthermore, these processes were removed in measuring execution times. The number of write accesses is the number of write accesses to DRAM and the NVM. When the language runtimes detect a write access to an object, the language runtimes examine the area of the target object. Next, the language runtimes increment the appropriate counter to record the number of write accesses. These counters are independent of the processing involved in our proposed method.

Figure 6 shows data regarding the average heap sizes. The data presented in this figure shows how our MRW queue method reduced the average heap size as compared to the individual counting method. Our MRW queue method reduced the average heap size by 2.9%-14.3%.

Table I summarizes the maximum sizes of the heap. Here, the positive reduction rates show how our MRW queue method reduced the maximum heap size versus the individual counting method. In contrast, the two negative reduction rates reveal how our MRW queue method increased the maximum heap size as compared to the individual counting method. This result shows that our MRW queue method reduced the maximum heap size by 1.5%-18.1%, except for the eclipse and luindex benchmarks.

Table II summarizes the execution times. Here, the reduction rates are the differences between the individual counting method and our MRW method. The positive values show how our MRW method reduced the overhead as compared to the individual counting method. The negative values show how our MRW method increased the overhead versus the individual counting method. These results show that our MRW queue method was an improvement in terms of overhead versus the individual counting method except for the antlr, pmd, and xalan benchmarks. In these three benchmarks, our MRW queue method took longer time than the individual counting method. However, these increases are lower than 2.0%.

Tables III and IV summarize the number of write accesses to the NVM space and DRAM space. They also show the percentage of NVM write accesses versus the total write accesses to memory (i.e., the sum of NVM and DRAM write accesses). The impacts of our MRW queue method were almost the same as the individual counting method, with differences less than 1.0%.

V. DISCUSSION

The results of our experiments showed the two positive effects of our MRW queue method. First, we observed the reduction of the average heap size. Our MRW queue method reduced the average heap size of the individual counting method by 2.9%-14.3%. Second, we observed the reduction of overhead. Our MRW queue method reduced the overhead of the individual counting method by 0.7%-10.17%. There were several exceptions in which our MRW queue method did not

TABLE II. EXECUTION TIME

	individual count (sec)	MRW queue (sec)	reduction (%)
antlr	14.69	14.94	-1.70
bloat	67.71	67.21	0.74
eclipse	408.73	383.10	6.27
fop	9.49	9.39	1.05
hsqldb	32.92	30.47	7.44
jython	74.48	71.51	3.99
luindex	81.32	75.56	7.08
lusearch	125.13	112.40	10.17
pmd	67.85	69.15	-1.92
xalan	70.52	71.53	-1.43

TABLE III. THE NUMBER OF WRITE ACCESS (INDIVIDUAL)

benchmark	NVM	DRAM	SUM	NVM/SUM (%)
antlr	651,327	129,736,724	130,388,051	0.50
bloat	1,282,922	763,708,687	764,991,609	0.17
eclipse	6,358,229	3,130,744,369	3,137,102,598	0.20
fop	763,291	46,433,270	47,196,561	1.62
hsqldb	3,746,670	56,533,575	60,280,245	6.22
jython	1,376,039	627,148,272	628,524,311	0.22
luindex	1,522,836	882,005,274	883,528,110	0.17
lusearch	1,424,364	270,661,267	272,085,631	0.52
pmd	6,740,167	243,320,668	250,060,835	2.70
xalan	2,438,810	179,047,994	181,486,804	1.34

reduce overhead; however, the increase rate here was less than 2.0%.

In contrast, the results did not show any positive effects regarding the reduction of write accesses to the NVM. One of the causes here was that the ideal parameters of our MRW method are not clear. Given that our MRW method has several parameters to determine its own behavior, future work is required here. In particular, among these parameters, MRW_DEPTH, MRW_THRESHOLD, and MRW_INSERT_POINT are important. The combination of these three parameters has a great influence on the determination of the placement of objects. We must find the ideal combination of parameter to get more improvements.

VI. RELATED WORK

Previous studies regarding methods to manage allocation in NVM/DRAM hybrid memory exist. In [8], Dhiman et al. evaluated a method in which the OS manages page allocations on either DRAM or the NVM based on characteristics of write accesses acquired from hardware. In [6], Zhang and Li evaluated a method in which the OS manages page allocations based on the frequency of page modifications. In both of these studies, the authors discussed management at the OS level. In contrast, in our present research, we discussed management at the language runtime level. Thus, our research is different than [8] and [6]. Moreover, this research differs in that our proposed method manages allocations in smaller units than methods of [8] and [6].

VII. CONCLUSION

In [9], Nakagawa and Oikawa proposed an NVM/DRAM hybrid memory management method called the individual counting method at the programming language runtime level. In this research, we described problems of the individual counting method. And to resolve these problems, we designed our MRW queue method to manage write access characteristics.

TABLE IV. THE NUMBER OF WRITE ACCESS (MRW)

benchmark	NVM	DRAM	SUM	NVM/SUM (%)
antlr	705,700	130,922,580	131,628,280	0.54
bloat	1,346,061	750,853,588	752,199,649	0.18
eclipse	8,309,760	3,111,572,112	3,119,881,872	0.27
fop	771,835	48,212,436	48,984,271	1.58
hsqldb	3,863,591	56,156,799	60,020,390	6.44
jython	1,172,710	625,019,017	626,191,727	0.19
luindex	1,323,795	869,561,219	870,885,014	0.15
lusearch	1,940,117	272,260,180	274,200,297	0.71
pmd	6,271,485	243,359,543	249,631,028	2.51
xalan	4,156,043	180,894,649	185,050,692	2.25

Results of our experiments showed that our MRW queue method had two positive effects as compared to the individual counting method. First, we reduced overhead. Second, we improved memory efficiency. In contrast, our MRW queue method did not show any positive influence on the reduction of write accesses to the NVM.

As mentioned in Section V, the ideal values of the MRW_DEPTH, MRW_THRESHOLD, and MRW_INSERT_POINT parameters are not clear. Therefore, our future work includes experiments with a variety of parameters to determine the ideal parameters.

ACKNOWLEDGMENT

This work was supported by Grant-in-Aid for JSPS Fellows Grant Number 261818.

REFERENCES

[1] D. Apalkov, A. Khvalkovskiy, S. Watts *et al.*, "Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM)," *J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, pp. 13:1–13:35, 2013.

[2] S. Lai, "Current status of the phase change memory and its future," in *IEDM '03 Technical Digest*, 2003, pp. 10.1.1–10.1.4.

[3] W. W. Zhuang, W. Pan, B. D. Ulrich *et al.*, "Novel colossal magnetoresistive thin film nonvolatile resistance random access memory (RRAM)," in *Proc. of IEDM '02*, 2002, pp. 193–196.

[4] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable Dram Alternative," in *Proc. of ISCA' 09*, 2009, pp. 2–13.

[5] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *Proc. of ISCA '09*. ACM, 2009, pp. 24–33.

[6] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," in *Proc. of PACT '09*. IEEE, 2009, pp. 101–112.

[7] P. Zhou, B. Zhao, J. Yang *et al.*, "A durable and energy efficient main memory using phase change memory technology," in *Proc. of ISCA' 09*, vol. 37, no. 3, Jun. 2009, pp. 14–23.

[8] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *Proc. of DAC '09*. ACM, 2009, pp. 469–664.

[9] G. Nakagawa and S. Oikawa, "Language Runtime Support for NVM/DRAM Hybrid Main Memory," in *Proc. of COOL Chips '14*, 2014, pp. 1–3.

[10] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[11] B. Alpern, C. R. Attanasio, J. J. Barton *et al.*, "The Jalapeno Virtual Machine," *IBM Syst. J.*, vol. 39, no. 1, pp. 211–238, 2000.

[12] S. M. Blackburn, R. Garner, C. Hoffman *et al.*, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Proc. of OOPSLA '06*. ACM Press, 2006, pp. 169–190.