Nonvolatile Main Memory Aware Garbage Collection in High-Level Language Virtual Machine

Chen Pan[1], Mimi Xie[1], Chengmo Yang[2], Zili Shao[3], and Jingtong Hu[1]
[1]School of Electrical and Computer Engineering, Oklahoma State University, Stillwater, OK, 74078, USA.
[2]Department of Electrical and Computer Engineering, University of Delaware, Newark, DE, 19716, USA.
[3]Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong.
{chen.pan,mimix,jthu}@okstate.edu, chengmo@udel.edu, cszlshao@comp.polyu.edu.hk

ABSTRACT

Non-volatile memories (NVMs) such as Phase Change Memory (PCM) have been considered as promising candidates of next generation main memory for embedded systems due to their attractive features. These features include low power, high density, and better scalability. However, most existing NVMs suffer from two drawbacks, namely, limited write endurance and expensive write operation in terms of both time and energy. These problems are worsen when modern high-level languages employ virtual machine with garbage collector that generates a large amount of extra writes on non-volatile main memory. To tackle this challenge, this paper proposes three techniques: Living Objects Remapping ($LORE$), Dead Object Stamping (DOS), and Smart Wiping with Maximum Likelihood Estimation ($SMILE$) to reduce the unnecessary writes when garbage collector handles objects. The experimental results show that the proposed techniques not only significantly reduce the writes during each garbage collection cycle but also greatly improve the performance of virtual machine.

1. INTRODUCTION

Non-volatile memories (NVMs) such as Resistive Random Access Memory (RRAM) [22, 21, 9] and Phase Change Memory (PCM) [11, 13, 19, 23], have many attractive features for embedded systems to employ them as main memory. Those features include ultra low leakage power, high-density, better scalability, non-volatility, and high immunity to soft errors, which enable embedded systems to achieve longer battery life, large memory capacity, and better reliability. However, NVMs also suffer from two drawbacks: limited write endurance and expensive write operation in terms of time and energy.

These problems will become even worse when high-level programming language are used to develop embedded applications [2, 12, 16]. High-level languages, such as Java and Python, enable fast application development. However, these programming languages need underlying High-Level Language Virtual Machine (HLVM) [20, 4] support. H-LVMs ease the application development by taking care of the runtime memory management. One of the main functions of HLVMs is garbage collection (GC) [10, 17, 5], which is responsible for recycling unused heap memory to reduce both memory fragmentation and the risk of triggering "out of memory" error. During GC, memory contents in the heap will be moved frequently and a large number of writes will be generated on the main memory. Without any optimization, the massive extra writes generated by the HLVMs will worsen the situation of NVMs when they are adopted as main memory for embedded systems.

To avoid this undesirable situation, this paper focuses on reducing the number of writes on non-volatile main memory (NVMM) generated by the garbage collector in HLVM. To the best of our knowledge, this is the first work that considers the effect of HLVM on NVMM due to the extra writes caused by GC. In particular, we propose *NVM-aware GC*, a non-volatile main memory aware garbage collection, composed of three techniques that can be incorporated in high level language virtual machine to improve its NVMs friendliness. These three techniques are:

1) A Living Objects Remapping ($LORE$) algorithm that eliminates the write operations on NVMM when GC moves objects;
2) A Dead Objects Stamping (DOS) algorithm that reduces write operations by reserving the unused heap memory instead of releasing it to the operating system (OS);
3) A Smart Wiping with Maximum Likelihood Estimation ($SMILE$) algorithm that estimates the amount of memory space needed to be released back to OS based on maximum likelihood estimation. It balances the NVMM lifetime extension and system performance overhead.

Out work is based on the popular Generational Mark and Sweep [10] garbage collection. However, the proposed techniques can also be easily extended to other GCs. Experimental studies show that the proposed *NVM-aware GC* generates 40% less write operations compared with the traditional Generation Mark_Sweep, thus making it more suitable for NVMM with constrained lifetime.

The rest of the paper is organized as follows. Section 2 gives a brief introduction of the background and related work. Section 3 illustrates the main ideas with a motivational example. In Section 4, the three proposed techniques $LORE$, DOS, and $SMILE$ are introduced one by one. Experimental results are presented in Section 5. Finally, Sec-

.

978-1-4673-8079-9/15/$31.00 ©2015 IEEE

tion 6 concludes this work.

2. BACKGROUND AND RELATED WORK

Recent advances in Non-Volatile Memory (NVM) [3, 23, 7] technologies such as flash memory [3, 7], Phase Change Memory (PCM), Spin-Transfer Torque Random Access Memory (STT-RAM) [15, 24, 14], and Ferroelectric Random Access Memory (FeRAM) [18, 8] have attracted great research interests due to their promising features. These features include high density, power-economy, low-cost, non-volatility properties, high immunity to soft errors, etc. As traditional CMOS-based SRAM and DRAM are experiencing high leakage power and have scalability issues, these NVMs become viable candidates for traditional DRAM and SRAM replacements.

However, the deployment of NVMs is not trivial. Comparing with SRAM or DRAM, almost all of the NVMs suffer from two major drawbacks. First, they have limited endurance compared with their counterpart. Second, the write operation normally consumes more energy and takes longer time to complete than the read operation. These drawbacks become even worse when softwares, such as HLVM, generate a large number of writes on NVMs.

HLVMs are platforms where high-level programming languages (HLLs) are implemented. HLVM acts as a single program on OS. Each time when HLVM runs, it is alternating between two types of execution cycles. One is program running cycle and the other is GC cycle. In program running cycle, objects are created. If objects occupy too much heap space, GC will be triggered. The program execution will be hung up and HLVM moves into GC cycle. GC, a key component of HLVM, plays a major role in maintaining HLVM's efficiency. Generally, GC is responsible for releasing memory space of the dead objects and moving the existing objects to reduce memory fragmentation to improve software efficiency.

Several different GC have been developed, including Mark-Sweep, Mark-Compact, and Generational-MS. Mark-Sweep first marks dead objects and then sweep them at the end of GC cycle. As shown in Figure 1, Mark-Sweep gradually generates a large amount of memory fragmentation and slows down the program as a result. To tackle this limitation, Mark-Compact was proposed to compact the memory space of the living objects together for better management. The mechanism of Mark-Compact is illustrated in Figure 2. While Mark-Compact eliminates memory fragmentation, it unfortunately introduces a lot of extra write operations for moving objects, thus affecting GC performance.

This paper focuses on Generational-MS, a hybrid GC mechanism of Mark-Sweep and Generational, which is popular in modern high-level languages such as Java. Generational-MS divides memory space into several disjoint regions (denoted as *generations*). This paper specifically considers a two-generation GC wherein two regions, namely *Nursery* and *Mature* are employed. The Nursery region is reserved for new objects, while the Mature region are used to store old generation objects. The age of the objects depends on the number of GC cycles which they have survived. When Nursery becomes full, GC-Nursery will be triggered over this area. This procedure wipes out dead objects and promotes objects that lived long enough into the Mature region. Similarly, when Mature becomes full, GC-Mature will be triggered to wipe out the dead objects among the whole Mature space. Overall, by adopting this two-region organization, Generational-MS reduces memory fragmentation and requires fewer object movement (only across region) than Mark-Compact.

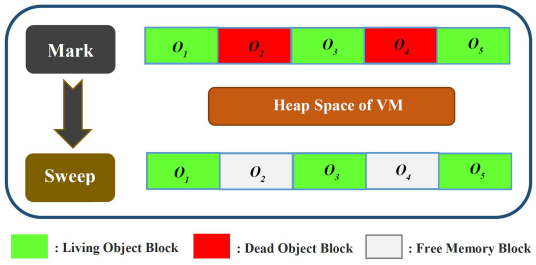

Figure 1: Mark-Sweep Garbage Collection

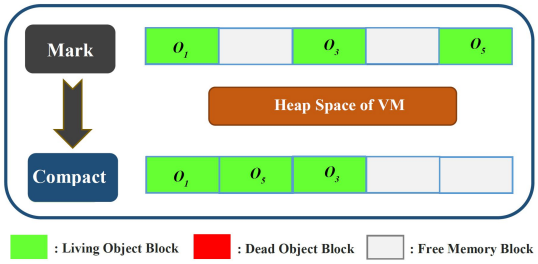

Figure 2: Mark-Compact Garbage Collection

To summarize, most GC schemes introduce extra memory writes to move objects, in order to reduce memory fragmentation. In most cases a program has many temporary objects which dies quickly. This will cause GC to be triggered frequently, engendering a large number of write operations to the heap region. These extra writes not only degrade GC performance, but also hurt the lifetime of NVM-based main memory. This motivates us to develop a NVM-aware GC, aiming at reducing unnecessary writes on NVMM during each GC cycle.

3. MOTIVATIONAL EXAMPLE

In this section, a motivational example is used to show that garbage collection in HLVM inevitably triggers extra write operations, which will affect the lifetime of NVM-based Main Memory.

For illustration purpose, in this example the heap is divided into two regions, namely, Mature and Nursery. Each region has two memory blocks. These four memory blocks will be mapped to four physical memory blocks. Each newly created object has the size of one memory block and will first be stored in Nursery. Once an object in the Nursery region has survived two GC cycles, it will be moved to the Mature region at the end of the 2^{nd} GC cycle. In this example, four objects, namely, O_1, O_2, O_3, and O_4, are generated sequentially. Whenever the Nursery or the Mature region is full,

GC will be triggered. Figure 3 shows the memory activities both on heap and in its mapped physical memory during each GC cycle. Each time a object survives from GC, its age value will be increased by one, which is shown as $O_i(*)$ in the figure.

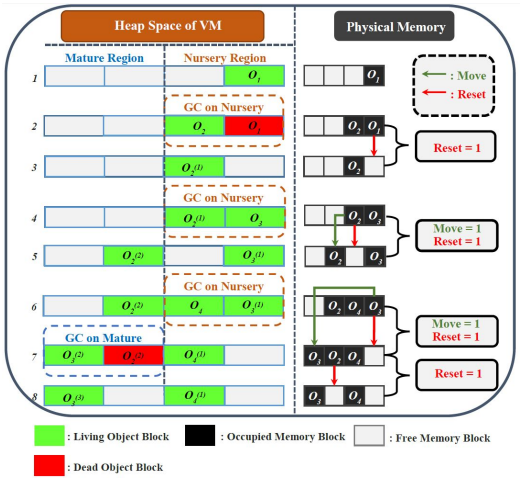

Figure 3: Garbage Collection Produce Writes on Physical Memory

When O_1 is generated, it is placed in Nursery. After O_2 is generated at step 2, the Nursery Region is full. Thus, GC is triggered on this region. Assuming that O_1 is now a dead object, GC will clean the memory space of O_1 by writing this whole block with initial values. We call it a "block-write". Here, cleaning the data before releasing the block can both protect the sensitive information of the program and reduce the risk of other programs mistakenly using uncleaned data. After O_1 being removed, object O_3 is generated in step 4, which makes Nursery full and GC is triggered again. At this time there is no dead object to reset. Yet O_2 now has the age value of one. Although it is still alive, GC will move O_2 into Mature. It first copies O_2 into its new destination, which requires one block-write, and then resets the memory block previous occupied by O_2 in Nursery, which takes another block-write. Similarly, at step 6, the moving of O_3 takes two block-writes. Finally, the Mature region is full and O_2 is dead at step 7. Hence, another block-write is required to release O_2 and reset its physical memory. Overall, this small example includes six total block-writes on NVMM, triggered by GC. Since each block could be 512 bytes or even larger, these GC activities incur considerable adverse effect on the NVMM main memory.

From the analysis above, one can see that these writes mainly come from object moving and resetting. However, these writes are unnecessary and can be avoided through careful address remapping and memory preserving. First, during object moving, if OS can directly remap the physical address from the original virtual address to the destination virtual address in the page table, there will be no physical data movement. This will, in this example, reduce all the block-writes due to object moving (two in total). Second, during object resetting, if GC reserves the memory space of the dead object without resetting it, and then directly overwrites it with newly generated objects, then one block-write can be avoided.

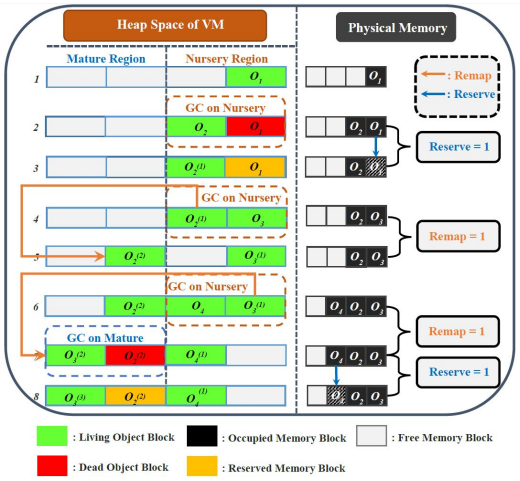

Figure 4: OS Controlled Garbage Collector to Avoid Unnecessary Writes on Physical Memory

Figure 4 shows the example of GC when the proposed address remapping and memory preserving is applied. When GC is triggered at step 2, the memory space of the dead object O_1 will be reserved until been overwritten by O_3 at step 4. This can reduce one block-write on physical memory. For the GC at step 4, O_2 will be remapped to Mature region (through changing logical-to-physical page mapping) instead of being physically moved there. Thus, two block-writes on physical memory are avoided. Similarly, the remapping of O_3 at step 6 will save two block-writes. Finally, at step 7 O_2 will be reserved to be overwritten by new objects, which eliminates one block-write. In this way, all the six extra block-writes in Figure 3 can be totally avoided in Figure 4.

Although remapping and reserving improve the endurance of NVMM, they may adversely influence the performance of other programs. Specifically, memory reserving can be achieved by forcing the virtual machine to reserve a large amount of memory space. However, the performance of other programs that run in parallel with the VM will also likely be affected due to the lack of memory.

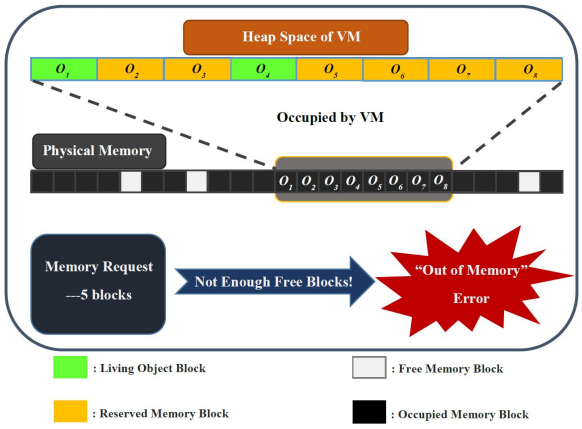

Figure 5: Potential Interference Between Programs

The potential interference between programs is illustrated in Figure 5. Assuming all the dead object space has been

reserved by GC, there are only three free blocks available in the physical memory and the others are occupied by current running programs including the VM. In this case, memory request of five free blocks by a current running program cannot be served, resulting in a "out of memory" error. This potential degradation in Quality-of-Service can be avoided, if the VM in this example releases two more reserved blocks. This motivates the development of an algorithm that takes into consideration the performance of both the VM and the other running programs, so as to determine the amount of reserved heap space to be released. As shown in Figure 6, if GC releases three reserved blocks, there will be a total of six free blocks available in physical memory. In this case, the memory request of five free blocks can be served smoothly. This will greatly improve system stability and speed compared with the situation shown in Figure 5.

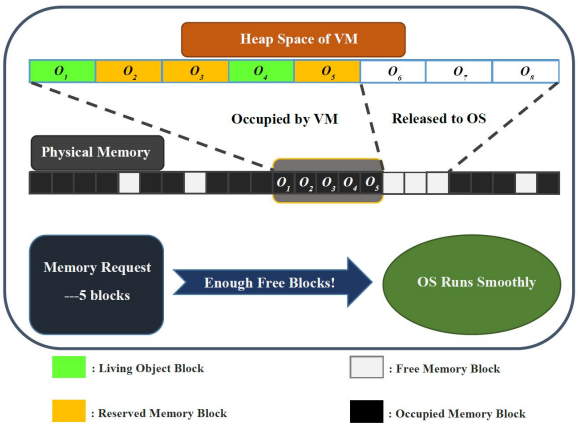

Figure 6: Release Heap Space by Considering Other Programs

Considering the aforementioned problems, three techniques will be proposed in this paper, with OS support to accomplish address remapping, memory reserving, as well as deciding the amount of heap space to be released after each GC cycle.

4. ALGORITHM

In this section, we will present *NVM-aware GC*, an NVM-M aware garbage collection algorithm, composed of three techniques that reduce the number of writes on NVMM.

4.1 NVM-aware GC Overview

The goal of the proposed *NVM-aware GC* algorithm is to minimize both the number of writes and the impact on system performance. It is executed during each GC cycle. Algorithm 4.1 shows the inputs at the beginning of each cycle. Inputs include moving object set MO_set, dead object set DO_set, page table PTB, current GC cycle index N, and an estimation of the required heap space $\hat{X}^N_{MLE}$ for current GC cycle. With these inputs, *NVM-aware GC* will fetch necessary information to execute the $LORE$, DOS, and $SMILE$ algorithms one by one. Functions of these three algorithms are listed below.

1) $LORE$ manages living objects to completely eliminate writes on NVMM during their movement;

2) DOS stamps dead objects, instead of wiping them out of main memory;

Algorithm 4.1 NVM-aware GC

Input: moving objects set MO_set, dead objects set DO_set, page table PTB, current GC cycle index N, initial size of occupied heap size X^N_{OCP} at the beginning of the N^{th} running cycle, and estimation of the required heap size $\hat{X}^N_{MLE}$ for current GC cycle

Output: Minimize both the writes on NVMM during the N^{th} GC Cycle and impact on system performance

1: **for** each object M_i in moving object set MO_set **do**
2: M^i_{Cur} ←current virtual addresses of object M^i;
3: M^i_{Des} ←destination virtual addresses of object M^i;
4: **end for**
5: **for** each dead object D_i in dead object set DO_set **do**
6: D^i_{Cur} ←current virtual addresses of object D^i;
7: **end for**
8: **for** each page table entry PTE^k in page table entries PTB **do**
9: PTE^k_{PHY} ← physical address of the k^{th} page table entry;
10: PTE^k_{VRT} ← virtual address of the k^{th} page table entry;
11: PTE^k_{STP} ← the stamp of the k^{th} page table entry;
12: **end for**
13: page table backup $PTB_{bak} \leftarrow PTB$;
14: ***LORE***(PTB, MO_set); /* Section 4.2 */
15: ***DOS***(PTB, DO_set); /* Section 4.3 */
16: ***SMILE***$(PTB, PTB_{bak}, N, X^N_{OCP}, \hat{X}^N_{MLE})$; /* Section 4.4 */
17: **return** PTB, X^{N+1}_{OCP}, and $\hat{X}^{N+1}_{MLE}$

3) In order to balance the system performance and NVM-M lifetime extension, memory space of some stamped objects will be released to OS. $SMILE$ will determine the number of stamped objects that need to be released.

These three algorithms are shown in Section 4.2, 4.3, and 4.4 respectively. Details of information required by these three sub-algorithms are shown in line 1-13 of Algorithm 4.1. Upon completion of these three algorithms, *NVM-aware GC* will save values of the latest page table PTB, initial size of occupied heap size X^{N+1}_{OCP} at the beginning of the $(N+1)^{th}$ running cycle, and the estimation of the required memory size $\hat{X}^{N+1}_{MLE}$ for the next GC cycle. With these three steps the number of writes on NVMM during each GC cycle can be significantly reduced with small system overhead.

4.2 Living Objects Remapping (LORE)

In this subsection, the Living Objects Remapping ($LORE$) algorithm will be presented. The goal of $LORE$ is to avoid unnecessary writes on NVMM when moving objects. For the garbage collection in $HLVM$, object moving happens under two circumstances. One is to copy objects from nursery region into mature region, while the other is to compact the scattered objects together in order to create large continuous space and reduce memory fragmentation. These object moving will trigger in writes on physical memory. To eliminate these unnecessary writes, the idea of $LORE$ is to inform OS to remap the physical address of the object from its original virtual address to the destination virtual address.

Details of $LORE$ are shown in Algorithm 4.2. During a GC cycle, $LORE$ is invoked when there are objects that

Algorithm 4.2 Living Objects Remapping (***LORE***)

Input: moving objects set MO_set, page table PTB
Output: remapped page table PTB

```
1: for each object M_i in MO_set do
2:     for each current address M^i_Cur[j] do
3:         for each page table entry PTE^k in PTB do
4:             if PTE^k_VRT = M^i_Cur[j] then
5:                 PTE^k_VRT ← M^i_Des[j];
6:             end if
7:         end for
8:     end for
9: end for
10: return PTB
```

Algorithm 4.3 Dead Objects Stamping (***DOS***)

Input: page table PTB, dead objects set DO_set
Output: page table entries with stamps PTE

```
1: for each object D_i in DO_set do
2:     for each current address D^i_Cur[j] do
3:         for each page table entry PTE^k in PTB do
4:             if D^i_Cur[j] = PTE^k_VRT then
5:                 stamp PTE^k with PTE^k_STP = 1;
6:             end if
7:         end for
8:     end for
9: end for
10: return PTB
```

need to be moved. The inputs include moving object set MO_set and page table PTB. In MO_set, each moving object is a 3-tuple element $M_i = \langle i, Cur, Des\rangle$, with i representing object ID and Cur and Des respectively representing M_i's current address and destination address in the virtual address space. Each PTB entry is also a 3-tuple element $PTE = \{\langle PHY, VRT, STP\rangle\}$ containing mapping information, where PHY and VRT represent the physical and virtual addresses respectively. Here, STP will be used by other components, which will be discussed later. Information regarding the objects that need to be moved will be provided by other key components of GC, e.g. reference counting and object tracing. Based on such information, $LORE$ will inform OS to change the mapping of the objects in the page table so that the physical addresses of the objects will remap from Cur to Des. This is shown in lines 4-6.

4.3 Dead Objects Stamping (DOS)

The Dead Objects Stamping (DOS) algorithm is proposed to stamp the dead objects on page table. During a traditional GC cycle, for security concerns, the contents of all the dead objects on physical memory have to be reset before their heap space is released to the OS at the end of each GC cycle. The reset procedure writes an initial value to the dead objects' heap space, which degrades system performance and NVMM's lifetime. This situation will become even worse when program generates a large amount of temporary objects which die quickly and eventually are overwritten with the initial value. To overcome this drawback, *NVM-aware GC* will not immediately reset and release these dead objects's memory space back to OS at the end of each GC cycle. Instead, *NVM-aware GC* will let DOS to stamp the dead objects. After that, the memory space of these stamped objects will be analyzed by the $SMILE$ algorithm, which will consider both the number of writes on NVMM and impact on system performance to determine how much memory space will be released back to OS.

Details of DOS are shown in Algorithm 4.3. Each time after GC identifies the dead objects, DOS will be executed to notify OS addresses of the dead objects. In order to do so, the page table PTB is extended with a new field called Stamp (STP). DOS informs OS by setting the stamp (STP) to $\widehat{Dead}$ in page table entry of each detected dead objects. This process is shown in lines 4 - 6. To reduce potential unnecessary writes on NVMM, memory space of these stamped dead objects will not be reset and released. Instead, they will remain intact. When a program running in VM needs space to store newly created objects, these stamped region will be allocated to it. Thus, the reset is avoided, which reduces the writes on NVMM. At the same time, OS does not need to allocate new physical memory space for these new objects, which also improves program efficiency.

One thing to note here is that at the beginning of each GC cycle, *NVM-aware GC* will copy page table PTB into a backup PTB_{bak} before changes are made to PTB by $LORE$ and DOS during current GC cycle. PTB_{bak} will be used by the $SMILE$ algorithm to identify how much memory space was actually required during the previous GC cycle. The details will be provided in the following subsection.

4.4 Smart Wiping with Maximum Likelihood Estimation (SMILE)

Since VM keeps all those dead objects stamped by DOS and does not release them to the OS, both VM efficiency and the number of writes on NVMM is reduced. However, occupying a large amount of memory space will also affect the performance of other programs running in parallel with VM. To prevent those programs from suffering from memory starving, it is necessary to find a balance between reducing the number of writes on NVMM and ensuring other programs' performance. To do so, one needs to determine the amount of memory space that the VM needs to release back to the OS. However, determining the exact amount is difficult since the exact number of newly generated objects between two consecutive GC cycles is unknown.

To solve the dilemma, the Smart Wiping with Maximum Likelihood Estimation ($SMILE$) algorithm is proposed to dynamically make a good estimation of the appropriate amount of reserved heap space for VM, so as to balance the number of writes on NVMM and the system performance degradation. The $SMILE$ algorithm is based on Maximum Likelihood Estimation (MLE) and is shown in Algorithm 4.4.

The main idea of $SMILE$ is to estimate how much heap space is needed by VM in the next running cycle based on MLE so that GC can release proper amount of space back to OS for other programs to use. The inputs include page table PTB, PTB_{bak}, number of GC cycles N, initial size of occupied heap size X^N_{OCP} at the beginning of the N^{th} running cycle, and maximum likelihood estimation of the required heap space $\hat{X}^N_{MLE}$ for the N^{th} running cycle. $\hat{X}^0_{MLE}$ is initialized to the default heap size during VM's initialization.

The initial estimation of heap space usage $\hat{X}^{N+1}_{MLE}$ for the $(N+1)^{th}$ cycle is based on MLE, which is computed by

Algorithm 4.4 Smart Wiping with Maximum Likelihood Estimation (***SMILE***).

Input: page table PTB, backup of PTB before GC PTB_{bak}, number of GC cycles N, initial size of occupied heap size X^N_{OCP} at the beginning of the N^{th} running cycle, and maximum likelihood estimation of the required heap space $\hat{X}^N_{MLE}$ for the N^{th} running cycle

Output: PTB, X^{N+1}_{OCP}, and $\hat{X}^{N+1}_{MLE}$

1: $TH^{VM}_{IMP} \leftarrow$ the threshold size of new heap space that VM applied from OS during the N^{th} running cycle without triggering significant impact on VM's performance;
2: $TH^{SYS}_{IMP} \leftarrow$ the threshold size of heap space of dead object that VM reserved during the N^{th} running cycle without triggering significant impact on system's performance;
3: $X_{Req} \leftarrow$the total size of heap space actually required in the N^{th} running cycle based on $STP_{bak} = \widehat{Live}$ in PTB_{bak}
4: $X_D \leftarrow$the size of heap memory space occupied by dead objects after execution of DOS based on $STP = \widehat{Dead}$ in PTB
5: $X_L \leftarrow$the size of heap memory space occupied by living objects after execution of DOS based on $STP = \widehat{Live}$ in PTB
6: $X_{Offset} \leftarrow$the size of heap memory space of dead objects which is occupied by VM yet it is not used to allocate new objects during the last running cycle based on $X^N_{OCP} - X_{Req}$
7: $\hat{X}^{N+1}_{MLE} = \frac{X_{Req}+(N-1)\hat{X}^N_{MLE}}{N}$
8: **if** $X_{Offset} > 0$ *and* $X_{Offset} > TH^{SYS}_{IMP}$ **then**
9: $\quad X^{N+1}_{OCP} = \hat{X}^{N+1}_{MLE} - X_{Offset} + TH^{SYS}_{IMP}$;
10: **else if** $X_{Offset} < 0$ *and* $\mid X_{Offset} \mid > TH^{VM}_{IMP}$ **then**
11: $\quad X^{N+1}_{OCP} = \hat{X}^{N+1}_{MLE} - X_{Offset} - TH^{VM}_{IMP}$;
12: **end if**
13: **if** $X^{N+1}_{OCP} < X_D + X_L$ **then**
14: $\quad R = X_D + X_L - X^{N+1}_{OCP}$
15: $\quad$ **if** $R < X_D$ **then**
16: $\quad\quad$ wipe and release the memory space occupied by dead objects with the size of R;
17: $\quad\quad$ change the STP state of all the rest of dead objects into $STP = \widehat{Reserve}$;
18: $\quad$ **else**
19: $\quad\quad$ wipe and release all the memory space occupied by dead objects;
20: $\quad$ **end if**
21: $\quad$ update PTE
22: $\quad$ **return** PTB, X^{N+1}_{OCP}, and $\hat{X}^{N+1}_{MLE}$
23: **end if**

the equation shown in Line 7. Here, $\hat{X}^N_{MLE}$ represents the maximum likelihood estimation of the required memory size for the N^{th} running cycle. The estimation is based on the assumption that the required heap size follows Poisson Distribution. Since each GC is triggered independently and purely random, and the amount of required heap size X_{Req} dose not vary with time, the assumption is reasonable. Theorem 4.1 provides the rational of the estimation. However, in addition to MLE, we also want to take the accuracy of previous cycle's estimation into consideration. If we have overestimated heap memory usage during the previous cycle, we will subtract the overestimated amount from MLE. If we have underestimated the usage, we will add the underestimated amount.

In order to know how much VM has overestimated or underestimated, GC needs to know the actual used heap memory size X_{Req} and the actual occupied heap size X^N_{OCP} during the N^{th} running cycle. If X^N_{OCP} is larger, then VM has reserved too much. Otherwise, the VM has reserved too little. Since X^N_{OCP} is already known as one of the inputs, we only need to obtain X_{Req}. Line 3 obtains the value of X_{Req}. The value is obtained based on PTB_{bak} which contains the object states at the end of the N^{th} program running cycle. At that point, PTB_{bak} has not been altered by the DOS algorithm in current GC cycle. Hence, the total memory size of objects in PTB_{bak} with $STP_{bak} = \widehat{Live}$ equals to X_{Req}. Once X_{Req} is acquired, GC compares it with X^N_{OCP} to see how much heap size is overestimated or underestimated. If X^N_{OCP} is larger, we have overestimated X^N_{OCP}-X_{Req}. If X_{Req} is larger, we have underestimated X_{Req}-X^N_{OCP}.

After we know how much we have overestimated or underestimated, we also want to avoid unnecessary adjustment in the estimation. Therefore, we define two thresholds, TH^{VM}_{IMP} and TH^{SYS}_{IMP}, in line 1 and 2. TH^{VM}_{IMP} is the memory size that VM applied from OS during the N^{th} running cycle without triggering significant impact on VM's performance. We will add the underestimation amount only when it is larger TH^{VM}_{IMP}. TH^{SYS}_{IMP} is the memory size that VM reserved during the N^{th} running cycle without triggering significant impact on system's performance. We will subtract the overestimation amount only when it is larger TH^{VM}_{IMP}. These two thresholds are set empirically. The details of this computing process is given in line 8 to 12. This generates X^{N+1}_{OCP}, which is the memory size that VM is going to reserve for the $(N+1)^{th}$ program running cycle.

After determining the value of X^{N+1}_{OCP}, GC will compared it with current occupied heap, which is the summation of X_D and X_L. If $X^{N+1}_{OCP} < X_D + X_L$, we will release the over-occupied memory space to the OS. This process is shown from Line 13 - 20. When over-occupied memory space has been wiped out and released back to OS, any remaining memory space occupied dead objects will be stamped as $\widehat{Reserve}$ to indict that the memory is available for new allocation.

THEOREM 4.1. *Suppose the required memory space during each program running cycle follows Poisson Distribution. If GC has been trigger N times, the actual required memory space of the N^{th} program running cycle is X_{Req}, and the previous Maximum Likelihood Estimation of required space is $\hat{X}^N_{MLE}$, then the Maximum Likelihood Estimation of the required page for the $(N+1)^{th}$ program running cycle is*

$$\hat{X}^{N+1}_{MLE} = \frac{X_{Req} + (N-1)\hat{X}^N_{MLE}}{N}$$

PROOF. Suppose there is a random sample $X_1, X_2, \cdots, X_N$ of N independent observations of required memory space during N GC cycles. They follow Poisson Distribution $X \sim \pi(\lambda)$ with unknown parameter vector λ. Since the Probability Mass Function of $X \sim \pi(\lambda)$ is:

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

Then the Maximum Likelihood Function with respect to pa-

rameter vector λ is:

$$L(\lambda) = \prod_{i=1}^{N} P_{x_i}(X_i|\lambda)$$

Take natural logarithm for $L(\lambda)$, then we have

$$\begin{aligned} l(\lambda) =& \ln \prod_{i=1}^{N} \frac{e^{-\lambda}\lambda^{X_i}}{X_i!} \\ =& -N\lambda + (\sum_{i=1}^{N} X_i)\ln(\lambda) - \sum_{i=1}^{N} \ln(X_i!) \end{aligned}$$

Now, take the derivative of $l(\lambda)$ with respect to λ, and set the value to 0, we get:

$$\frac{d}{d\lambda} l(\lambda) = 0 \Leftrightarrow -N + \frac{1}{\lambda}\sum_{i=1}^{N} X_i = 0$$

Here, we get a stationary point of $\hat{\lambda} = \frac{1}{N}\sum_{i=1}^{N} X_i$. Now, we calculate the second order derivative of $l(\lambda)$ with respect to λ to verify whether $\hat{\lambda}$ is the maximum likelihood value.

$$\frac{d^2}{d\lambda^2} l(\lambda) = -\frac{1}{\lambda^2}\sum_{i=1}^{N} X_i < 0$$

Hence, we have the maximum likelihood estimation of λ based on previous N observations of X as

$$\hat{\lambda}_{MLE}^{N} = \frac{1}{N}\sum_{i=1}^{N} X_i$$

Since the expected value of $X \sim \pi(\lambda)$ is $E[X] = \lambda$, with the initial value of $X_{MLE}^{0} = X_0$ been given, the estimation goes with the following process.

$$\begin{aligned} X_0 \Rightarrow & \hat{\lambda}_{MLE}^{0} = \hat{X}_{MLE}^{1} \\ X_1 \Rightarrow & \hat{\lambda}_{MLE}^{1} = \hat{X}_{MLE}^{2} \\ & \vdots \\ X_N \Rightarrow & \hat{\lambda}_{MLE}^{N} = \hat{X}_{MLE}^{N+1} \end{aligned}$$

Therefore, the best estimation for the required memory page of the next running cycle would be $\hat{X}_{MLE}^{N+1} = \hat{\lambda}_{MLE}^{N} = \frac{1}{N}\sum_{i=1}^{N} X_i$. With $X_N = X_{Req}$ as the actual required memory space during the N^{th} running cycle, the Maximum Likelihood Estimation of the required memory space of the next running cycle is

$$\begin{aligned} \hat{X}_{MLE}^{N+1} = \hat{\lambda}_{MLE}^{N} =& \frac{1}{N}\sum_{i=1}^{N} X_i \\ =& \frac{1}{N}(X_{Req} + \sum_{i=1}^{N-1} X_i) \\ =& \frac{1}{N}(X_{Req} + (N-1)\hat{\lambda}_{MLE}^{N-1}) \\ =& \frac{X_{Req} + (N-1)\hat{X}_{MLE}^{N}}{N} \end{aligned}$$

This completes the Proof of the Theorem 4.1 □

The proposed techniques *LORE*, *DOS*, and *SMILE* are built upon the knowledge of dead objects and moving objects, which are provided by other key functions of GC such as Objects Tracing, Reference Counting (RC), etc. *NVM-aware GC* only changes the sweep procedure of the traditional garbage collections. The other key functions remain intact. Therefore, it can be easily integrated into widely used Mark and Sweep (MS) based garbage collectors.

5. EXPERIMENTS

In this section we will evaluate the effectiveness of the proposed *NVM-aware GC* in terms of number of writes, energy consumption, and write overhead. Experimental setup is presented in Section 5.1. The experimental results of the proposed algorithm are given in Section 5.2.

5.1 Experimental setup

The experiment is based on Jikes RVM [1], an open source Java virtual machine, which provides a flexible open testbed for experiments. Jikes RVM provides a variety of GC methods with extendable interfaces, which enable us to monitor the memory activities during each GC cycle. In the experiments, we use the *Generational MS* garbage collection. We record the memory operations triggered by GC under different benchmarks during the GC cycles. Ten benchmarks come with Jikes RVM are used.

Table 1: Experimental Setup

Component	Description
CPU	Frequency: 1.0 GHz
NVM Main Memory	Memory Size: 128 MB, Page Size: 4 KB Read Latency: 50 ns/page, Write Latency 150 ns/page Read Energy: 0.048 nJ/page, Write Energy: 1.775 nJ/page
Heap	Initial size: 1 MB, Nursery Factor: 0.15

Table 1 shows the experimental setups of both hardware and software parameters. PCM is used as the NVM in the simulator. The parameters are collected from NVSim [6]. The memory size is 128 MB and each page is with standard size of 4 KB. For heap, the initial size is 1 MB. The Nursery Factor is 0.15, which means that the Nursery Region takes up 15% of total size of heap. When *NVM-aware GC* executes, the threshhold TH_{IMP}^{VM} is set as 10% of initial heap size and TH_{IMP}^{SYS} is set as 5% of initial heap size. Once these thresholds have been reached across, changes on heap size will be made.

5.2 Experimental Results

In this section, a series of experiments are conducted to evaluate the performance of *NVM-aware GC* in terms of number of writes, energy efficient, delay overhead, and the influences on overall system performance. Table 2 first shows the number of writes on Nursery and Mature Regions.

Table 2 shows that the numbers of writes on both Nursery region and Mature region have been reduced by 47.52% and 57.76%, respectively. Totally, the average writes have been reduced almost by half compared with *Generational MS* garbage collector.

Figure 7 to 9 show how the real writes on NVMM are distributed with selected benchmarks. Figure 7 shows the comparison of *GenMS* and *NVM-aware GC* with respect to the number of writes on NVMM for the *Spawn* benchmark. As we can see when using the *GenMS* garbage collector,

Table 2: The # of Writes on Nursey and Mature Regions with Different GC Engines

Bench.	GenMS		NVM-aware GC		%Reduction		
	Nursery Region	Mature Region	Nursery Region	Mature Region	Nursery	Mature	Total
Spawn	79326	6916	41663	1771	47.48	74.39	49.64
ReferenceTypes	58921	1242	31201	1190	47.05	4.19	46.22
SpreadAlloc	29416	14436	16428	7517	44.15	47.93	45.31
FixedLive	7249	4733	3821	126	47.29	97.34	67.06
CyclicGarbage	10088	585	5192	348	48.53	40.51	48.12
AlignedLists	3654	2441	2137	1094	41.52	55.18	46.97
QuickSort	4764	1206	2271	131	52.32	89.14	59.74
Lists	3277	1570	1512	1233	53.86	21.46	43.37
Concurrent2	1821	157	819	99	55.03	36.94	53.59
OutOfMemory	810	744	617	54	23.83	92.74	56.82
Average	18435	4901	9675	2070	47.52	57.76	49.67

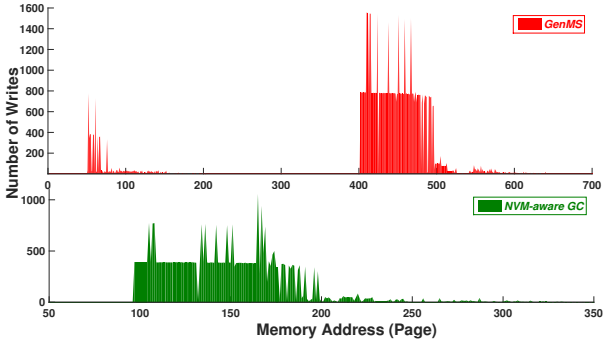

Figure 7: Comparison of number of writes between *GenMS* and *NVM-aware GC* with *Spawn* Benchmark

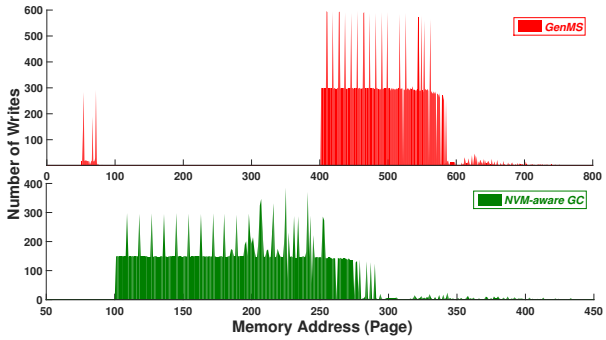

Figure 8: Comparison of number of writes between *GenMS* and *NVM-aware GC* with *ReferenceTypes* Benchmark

the Nursery region is mapped to the higher addresses which start from 400^{th} page. The Mature region is mapped to the lower address which starts at 50^{th} page. The writes on the mapped Nursery region in NVMM is significantly high, which is 79326 times as shown in Table 2. However, when applying *NVM-aware GC*, the writes are significantly reduced. This is mainly due to two reasons. First, in *NVM-aware GC*, there is no need to move the old objects from Nursery region into Mature region in physical memory. Instead, *NVM-aware GC* only changes the mapping information of the virtual address in page table. In *NVM-aware GC*, the writes on NVMM are accumulated together since the objects in Mature region only change the virtual addresses while the physical addresses remain intact. Second, the memory space occupied by dead objects in Nursery region will not be swept out during each GC cycle. Instead, it will be held and allocated to newly generated objects.

Figure 8 shows the comparison of *GenMS* and *NVM-aware GC* with respect to the number of writes on NVMM for the *ReferenceTypes* benchmark. Similarly, *NVM-aware GC* only changes the mapping of the old objects, so the old objects remain intact in physical memory. As we can see, there are only a few writes on NVMM where Mature region is mapped. This means that there are only a small portion of old objects and the majority of the objects are dead. Therefore, the *NVM-aware GC* only produces almost half of writes compared with *GenMS*.

Figure 9 shows the comparison of *GenMS* and *NVM-aware GC* with respect to the number of writes on NVMM

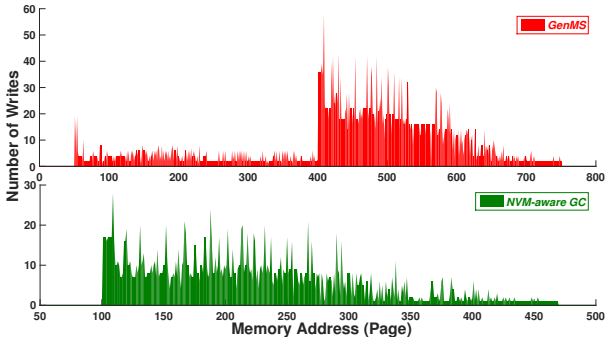

Figure 9: Comparison of number of writes between *GenMS* and *NVM-aware GC* with *QuickSort* Benchmark

for the *QuickSort* benchmark. As we can see the Mature region has overleaped with Nursery region physically. This is because, a lot of objects are old enough to be moved into Mature region and take a large space in the Mature region other then being swept out when they are young and dead. Since these objects' moving and sweeping can be avoided with remapping and space reserving, *NVM-aware GC* shows significant reduction in the number of writes on NVMM.

The experiments have revealed that the majority of objects die when they are young and in the Nursery region. Therefore, *NVM-aware GC* can be significantly helpful in reducing the number of writes due to the sweeping of the dead objects. Since the write operations on NVM are both en-

ergy and time consuming, *NVM-aware GC* can also reduce the energy consumption and improve system performance thanks to the reduction of memory activities. Table 3 and 4 present the experimental results of energy consumption and memory access latency in details.

Table 3: Memory Energy Consumption with Different GC Engines

Bench.	GenMS (μJ)		NVM-aware GC (μJ)	%Reduction
	write	read		
Spawn	153.081	0.332	77.095	49.75
ReferenceTypes	106.774	0.035	57.494	46.17
SpreadAlloc	77.849	0.693	42.502	45.88
FixedLive	21.093	0.225	6.614	68.98
CyclicGarbage	18.955	0.023	9.841	48.12
AlignedLists	10.823	0.12	5.735	47.56
QuickSort	10.597	0.072	4.265	60.02
Lists	8.079	0.073	4.872	40.17
Concurrent2	3.193	0.007	1.629	49.04
OutOfMemory	2.762	0.036	0.815	70.84
Average	41.316	0.161	21.086	49.16

Table 3 shows the energy consumption due to the memory operation on NVMM with *GenMS* and *NVM-aware GC* engines. For memory activities with *GenMS*, each time when a old object is moved from Nursery region to Mature region, two memory writes and one memory read will occur. This is because, the object will first be read into cache and then written to destination, which generates one read and one write. After the object being moved, the original copy will then be swept. This will produce another write. However, for *NVM-aware GC*, there is no writes being triggered during object moving as it remaps the addresses in page table. This not only reduces the the memory operations but also reduces the energy consumption. The only energy consumption with *NVM-aware GC* engine is from the new objects allocation. As we can see from Table 3, the energy has been significantly reduced with the average reduction of 49.16%.

Table 4: Memory Latency with Different GC Engines

Bench.	GenMS (μs)		NVM-aware GC (μs)	%Reduction
	write	read		
Spawn	12936	345.8	6515.1	50.95
ReferenceTypes	9023	36.8	4858.6	46.37
SpreadAlloc	6578	721.8	3591.8	50.8
FixedLive	1783	234.2	558.9	72.29
CyclicGarbage	1601	24.3	831.6	48.83
AlignedLists	914	122.1	484.6	53.23
QuickSort	896	75.3	360.4	62.87
Lists	662	76.5	411.8	45.69
Concurrent2	270	7.95	137.7	50.38
OutOfMemory	233	37.2	68.8	74.53
Average	3491.5	163.15	1781.9	51.31

As the writes on NVM are time consuming, *NVM-aware GC* can also reduce the memory latency as the memory activities are reduced. Table 4 shows the memory latency results in detail. From Table 4, we can see that due to the reduced memory writes and read on NVMM, the *NVM-aware GC* can dramatically reduced the time overhead to more than 50% on average. Therefore, with the reduction of writes on NVMM *NVM-aware GC* can improve the system performance.

Although *NVM-aware GC* can reduce memory writes, energy, and access latency on memory, *NVM-aware GC* also requires to reserve the memory space of the dead objects. This may degrade the stability of system performance, as other programs run at the same time may suffer from "out of memory" error when there is not enough memory space available for allocation. Therefore, we compare the changes of heap size for *NVM-aware GC* and *GenMS* in Figure 10 to observe the adjustment of heap size during multiple GC cycles.

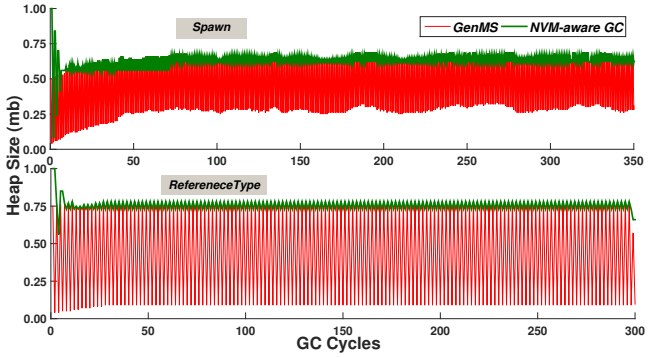

Figure 10: Comparison of Heap Size Changes between *GenMS* and *NVM-aware GC* with *Spawn* and *ReferenceTypes* Benchmarks

Figure 10 shows the results for two benchmarks: Spawn and ReferenceType. The results show that the heap sizes with *NVM-aware GC* are stabilized after a couple of GC cycles in both benchmarks. And these stable heap sizes do not exceed the maximum required heap size of each running cycle. Therefore, if *GenMS* can keep the system stable, *NVM-aware GC* can too. Therefore, *NVM-aware GC* is as table as *GenMS*.

For *GenMS*, the heap size changes with great fluctuation. This is because, each time when GC is triggered, dead objects will be swept and their occupied memory space will be return to OS at the end of the GC cycle. Therefore, VM needs to initiate memory requests of new space for new objects. This increases the burden for OS. In contrast, in *NVM-aware GC*, the heap space is prepared for the next running cycle at the end of current cycle. As the $SMILE$ algorithm dynamically predicts the required heap size of the next running cycle based on previous observation, the more times $SMILE$ observes, the more accurate the prediction will be.

6. CONCLUSION

In this paper, a non-volatile main memory aware garbage collector in HLVM is designed to reduce the write operations from garbage collection in HLVM. Existing garbage collectors have not considered about the limited write endurance of NVMM. Therefore, this paper proposes three techniques, namely, Living Objects Remapping ($LORE$), Dead Object Stamping (DOS), and Smart Wiping with Maximum Likelihood Estimation ($SMILE$) algorithm to reduce the unnecessary writes on NVMM. The experimental results show that the proposed *NVM-aware GC* can reduce nearly half of the write operations during each GC cycle compared with widely used *GenMS* garbage collection. This will significantly lower the energy consumption and latency due to reduction of memory activities. The drawback of *NVM-aware GC* is the reservation of the memory space of the dead objects, the

experimental results have shown that the reserved memory space can quickly reach to its stable state and will not exceed the maximum required heap space of each running cycle. Hence the proposed *NVM-aware GC* is as stable as existing GCs.

7. ACKNOWLEDGMENTS

This work is supported by NSF CNS-1464429, grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 152138/14E), and National Natural Science Foundation of China (Project 61272103 and Project 61373049).

8. REFERENCES

[1] `http://jikesrvm.org/`.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[3] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 181–192, 2009.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[5] G. Czajkowski and L. Daynàs. Multitasking without compromise: a virtual machine evolution. *ACM SIGPLAN Notices*, 47(4a):60–73, 2012.

[6] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(7):994–1007, 2012.

[7] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *MICRO '09*, pages 24–33, 2009.

[8] K. Hoya, D. Takashima, S. Shiratake, R. Ogiwara, T. Miyakawa, H. Shiga, S. M. Doumae, S. Ohtsuki, Y. Kumura, S. Shuto, et al. A 64-mb chain feram with quad bl architecture and 200 mb/s burst mode. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(12):1745–1752, 2010.

[9] D. Ielmini. Modeling the universal set/reset characteristics of bipolar rram by field- and temperature-driven filament growth. *Electron Devices, IEEE Transactions on*, 58(12):4309–4317, Dec 2011.

[10] R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, 1996.

[11] M. Joshi, W. Zhang, and T. Li. Mercury: A fast and energy-efficient multi-level cell based phase change memory system. In *HPCA '11*, pages 345–356, 2011.

[12] S. Katzen. High-level language. In *The Essential PIC18® Microcontroller*, pages 275–302. Springer, 2010.

[13] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 37(3):2–13, June 2009.

[14] J. Li, L. Shi, Q. Li, C. J. Xue, Y. Chen, Y. Xu, and W. Wang. Low-energy volatile stt-ram cache design using cache-coherence-enabled adaptive refresh. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 19:5, 2013.

[15] Q. Li, J. Li, L. Shi, C. J. Xue, and Y. He. Mac: migration-aware compilation for stt-ram based hybrid cache in embedded systems. In *ISLPED '12*, pages 351–356, 2012.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[17] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *ACM Sigplan Notices*, volume 45, pages 146–159. ACM, 2010.

[18] H. Shiga, D. Takashima, S. Shiratake, K. Hoya, T. Miyakawa, R. Ogiwara, R. Fukuda, R. Takizawa, K. Hatsuda, F. Matsuoka, et al. A 1.6 gb/s ddr2 128 mb chain feram with scalable octal bitline and sensing schemes. *Solid-State Circuits, IEEE Journal of*, 45(1):142–152, 2010.

[19] D.-J. Shin, S. K. Park, S. M. Kim, and K. H. Park. Adaptive page grouping for energy efficiency in hybrid pram-dram main memory. In *RACS '12*, pages 395–402, 2012.

[20] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[21] H.-S. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. Chen, and M.-J. Tsai. Metaĺcoxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, June 2012.

[22] C. Xu, X. Dong, N. Jouppi, and Y. Xie. Design implications of memristor-based rram cross-point structures. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[23] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. *SIGARCH Comput. Archit. News*, 37(3):14–23, June 2009.

[24] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Energy reduction for stt-ram using early write termination. In *ICCAD '09*, pages 264–268, 2009.