Automatic Code Conversion for Non-Volatile Memory

Jinsoo Yoo Yongjun Park Seongjin Lee* Youjip Won
Hanyang University, Seoul, Korea
Gyeongsang National University, Jinju, Korea*

ABSTRACT

Non-Volatile Memories (NVMs), such as Phase Change Memories (PCMs) and Resistive RAMs (ReRAMs), have been recently proposed as a main memory due to their higher capacity and low leakage power consumption compared to traditional DRAMs. In order to support the NVM-based systems, many software platforms are developed and they provide user-level programming interfaces. However, many existing applications are already written based on the conventional DRAM-based systems; thus, programmers have to rewrite or modify the code in order for the code to successfully run on NVM-based systems. In order to solve this problem, we introduce a code-conversion tool named a *Code Regenerator* that transforms applications that are originally designed for conventional operating systems using DRAM as a main memory into applications that runs on HEAPO which is a non-volatile memory based software platform. The code regenerator consists of code profiler and code generator. Among all dynamic and static memory objects of an application, code profiler profiles the code to find the objects that fit well into the characteristics of NVM. Based on the profiling result, code generator re-writes the target application code to exploit NVM through HEAPO programming interfaces based on the profiling result. In this paper, we demonstrate that applications transformed through code regenerator stably run on NVM platform without manual code modification. By allocating read-intensive memory objects to NVM, the regenerated applications reduce the energy consumption by up to 44% compared to that of the original applications.

CCS CONCEPTS

• **Software and its engineering → Source code generation**;

KEYWORDS

Non Volatile Memory, Compiler, Code Generator

ACM Reference Format:
Jinsoo Yoo, Yongjun Park, Seongjin Lee, and Youjip Won. 2018. Automatic Code Conversion for Non-Volatile Memory. In *SAC 2018: SAC 2018: Symposium on Applied Computing , April 9–13, 2018, Pau, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3167132.3167246

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167246

1 INTRODUCTION

Volatile memories such as DRAM have been widely used as main memory in all computer systems from embedded systems to supercomputers. While DRAM on those computer systems serves the main role of data keeping, recovery, and serialization, increasing the size of DRAM based main memory has reached its limit due to increasing cost (DRAM technology scaling slowdown) and power consumption (higher leakage currents) problems [19]. Data-intensive internet services provided by Facebook and YouTube require massive servers with an enormous amount of DRAM to increase its performance, but the aforementioned limitations of DRAM make it harder. Recent studies [1, 5] show that the energy consumption of main memory has become the main challenge for building a data center. Furthermore, additional data recovery schemes are required for a DRAM-based system due to the volatile property which is a source of performance degradation.

To solve the limitations, Non-volatile memories (NVMs) such as Flash, Spin-transfer Torque Magnetic RAM (STT-MRAM) [11], Ferroelectric RAM (FeRAM) [13], and Phase-change RAM (PRAM or PCM) [20] have been developed. Since these devices have favorable characteristics, the NVMs are proposed to replace the existing components of memory hierarchy from caches to secondary storages. For example, STT-MRAM is expected as a promising device to replace DRAM used for the main memory of computer systems because it has comparable read/write latencies and endurance to DRAM. PRAM is expected to be used for secondary storage instead of HDD or Flash. As NVMs (except Flash) are byte-addressable and persistent, it is expected to solve serialization overhead and high power consumption problem of DRAM. Lee et al. [17] showed that NVM can reduce the execution time of loading web pages and remove serialization overhead of the system by storing the whole data structure in NVM-based persistent secondary storage. Qureshi et al. [20] found that the hybrid memory systems consisting of DRAM and NVM can reduce the total energy consumption by 35% compared to the DRAM-only systems while increasing the performance up to 3×.

While NVM devices were in development, software community initiated efforts to efficiently support the NVM based hardwares. Many of the attempts to use non-volatile memory were focused on file systems [7, 15, 25], and persistent object programming interfaces [2]. A persistent object is a memory object that is kept permanently in a storage and accessed by process unless it is deleted from a secondary storage [22]. Software platforms supporting persistent objects provide several programming models and APIs, such as C language user-level programming APIs [8, 12, 23] and C++ language programming APIs [4], to manage them efficiently. For example, NVL-C [6] is an extension of the C programming model, which allows a programmer to explicitly declare variables to be allocated to an NVM.

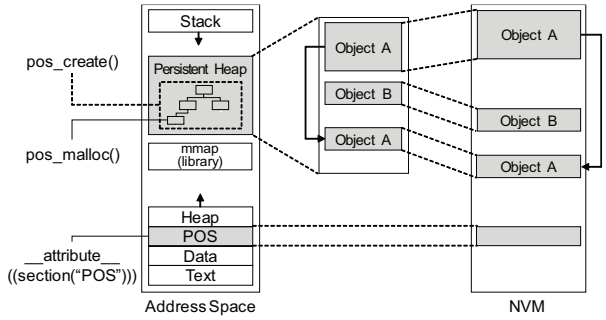

Figure 1. The Address Space Layout in HEAPO

To benefit from using the persistent object, applications should be designed properly using the programming models and APIs, but most applications are already designed to run on a conventional operating system that uses DRAM as a main memory. Since most existing programs do not utilize persistent objects, programmers need to modify application code to use them according to programming models and APIs for given NVM devices. However, field studies [18, 21] show that it is difficult, time-consuming, and also error-prone for programmers to manually program an application for NVM with those programming models and APIs. NVMOVE [3] automatically assigns user-defined types which are created or updated in initialization or recovery phase to an NVM. NVMOVE has limitations that it selects specific variable types rather than specific variables to be allocated to an NVM. As a result, even if a few variables need to be persisted, the entire type is allocated to an NVM. Thus, a new code-conversion framework is mandatory.

In an effort to minimize the code modification overhead, we propose an automatic code conversion framework called *Code Regenerator* to apply an NVM based programming model and its APIs to existing applications that do not assume non-volatile memory systems. The code regenerator consists of *code profiler* and *code generator*. First, the *code profiler* finds memory objects that can be mapped to NVMs by exploiting profiled information of target application. Then the *code generator* changes the program code to exploit NVM objects using the APIs automatically, and users do not need to know how to apply NVM-based programming models and APIs to existing applications. We demonstrate the effectiveness of the code regenerator by automatically transforming original application codes and running them on a target non-volatile memory system. We also present a new criterion for which memory objects should be allocated to an NVM. The code regenerator profiles the energy consumption of each memory object and determines the memory to which a memory object is allocated. As a result, the energy consumption of the regenerated susan benchmark [9] program is reduced by 44% compared to that of the original program. A well-known non-volatile memory system named HEAPO is used to the target system of the code regenerator.

This paper is organized as follows. Section 2 introduces HEAPO NVM-based system and the code profiler. Section 3 explains how our proposed code regenerator works and section 4 shows the experimental results of regenerated applications.

2 BACKGROUND

2.1 Heap-Based Persistent Object Store

HEAPO [12] is a software platform for NVMs to manage persistent objects allocated in a persistent heap on an NVM enabled system. In contrast to other memory-mapped file based software platforms for NVMs, HEAPO supports light-weight persistent heap; HEAPO removes redundancy between in-memory and on-disk metadata and system call overhead to synchronize with a file system. The architecture of HEAPO is designed for a system with DRAM and an NVM on the same system bus and manages one physical memory space for both memories. Persistent objects can be allocated in an NVM using APIs from HEAPO programming library.

Code 1. HEAPO Persistent Object Programming APIs

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "pos-lib.h" /* HEAPO library
4                                  header */
5  /* persitent variable */
6  int global __attribute__((section("POS")));
7
8  int main(int argc, void **argv){
9     {/* create or map persistent object */
10       if(pos_create("object_name") < 1){
11          if(pos_map("object_nmae") < 1){
12             return -1;
13          }
14       }
15    }
16
17    /* allocate memory within persistent
          object */
18    char* stmp = (char *)pos_malloc(
19                "object_name", reclen * 2);
20
21    {/* freeing memory according to the
          address */
22       if(stmp >= 0x5FFEF80000 &&
23                stmp < 0x7FFEF80000){
24          pos_free("object_name", stmp);
25       }else{
26          free(stmp);
27       }
28    }
29 }
```

The address of persistent heap in HEAPO is defined in the address range starting from `0x5FFEF8000000` to `0x7FFEF8000000` in x86 64 bit architectures, and the total size of the persistent heap is 32 TBytes. HEAPO uses an independent namespace different from a file system but persistent objects can be managed just like a file in a file system. Fig. 1 shows the address space layout of persistent objects and persistent variables, according to the APIs provided by HEAPO. As shown in Fig. 1, two persistent objects are created using the HEAPO APIs in the persistent heap. The memory chunk of each object is allocated in the NVM area using the API `pos_malloc()`. HEAPO provides a custom C library to manage persistent objects. Code 1 shows an example to illustrate the use of the APIs for a target NVM-based system. Brief descriptions of HEAPO library APIs and a keyword for persistent variables used in the code are as follows:

- `"pos-lib.h"`: the name of HEAPO library

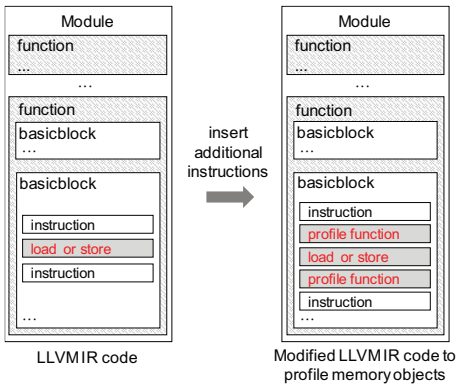

Figure 2. LLVM IR Code Modification by Profiler

- `__attribute__((section("POS")))`: persistent variable keyword
- `int pos_create(char *name)`: a function to create a persistent object called `name`.
- `int pos_map(char *name)`: a function to map a persistent object called `name` to the address space.
- `void* pos_malloc(char *name, unsigned long len)`: a function to allocate a block of `len` bytes of memory from a persistent object called `name`.
- `void pos_free(char *name, void *mem)`: a function to deallocate memory block allocated by `pos_malloc()`.

Algorithms used in `pos_malloc()` and `pos_free()` functions for memory allocation/deallocation, are similar to generic heap management algorithms for `malloc()` and `free()` functions in glibc. When allocating or freeing a memory space for a persistent object, the programmer can use `pos_malloc()` and `pos_free()` with the similar usage as `malloc()` and `free()`. The HEAPO interfaces differ from the existing memory allocation/deallocation interfaces in that `pos_malloc()` and `pos_free()` require a name of a persistent object as an additional argument.

In addition, HEAPO supports a special keyword for persistent variables: `__attribute__ ((section ("POS")))`. To declare a persistent variable, the keyword is used as a prefix or a suffix in addition to the general variable declaration. First, the persistent variables are stored in the POS segment in ELF of a HEAPO enabled system. Then the modified loader loads the POS segment into an NVM. As HEAPO APIs and persistent variable declaration are simple and straight forward, conversion of an application code into the code for NVM-based systems can be automated without any help of the compiler.

2.2 Code Profiler

Code profiler [10] uses front-end library of low-level virtual machine (LLVM) [16] compiler framework to modify intermediate representation (IR) code from programs. The front-end library provides programming APIs to modify LLVM IR code. To generate information of memory objects, profiler checks memory access instructions such as `load` and `store` in IR code. The profiler inserts profiling functions to the back and forth of each memory access instruction. During a profiling, the profile function counts how many the memory access instruction is called. Fig. 2 shows modified LLVM IR code to profile memory accesses in the code. Note that the program with profiling code executes more instructions and thus has longer execution time than that of the original program.

When IR code modification is done, code profiler executes the modified program and stores the profiling result to a file. The result includes the call count of memory access instructions for each memory object.

Table 1. Information Acquired by Code Profiler

Object Type	Information
Global Variable	File name, Variable name, Line number
Static Variable	File name, Variable name, Line number
Heap	File name, Function name, Line number

3 CODE REGENERATOR

3.1 Design

Code regenerator consists of a code profiler and a code generator. The role of the code profiler is to select memory objects that can be mapped into NVM from a target application by injecting profiling code in IR level. The code profiler informs the code generator memory object candidates to be mapped in an NVM. Using the objects information the code generator converts the target application into a NVM-based program.

To generate the modified code with persistent object programming APIs, the code generator needs information of memory objects which can be mapped to NVMs in the original program. Processes generally manage memory objects in stack, heap, and data (bss) segment. Among these segments, the code profiler considers memory objects in the heap and data segment as candidates for allocating in NVM. Memory objects in stack segment are not considered to be allocated in an NVM because the lifetime of the variables are too short to be beneficial when they are allocated in an NVM. We modified the code profiler to calculate the energy consumption of each variable in the heap and data segment when they are stored in DRAM or an NVM. The code profiler calculates the energy consumption by considering the number of memory accesses for each variable and the read/write energy consumption of DRAM and NVM. The parameters used for calculating the energy consumption of read and write operation of the DRAM is 0.7nJ, and 0.4nJ and 2.3nJ for read and write operations on an NVM, respectively [24]. Since the power consumption of NVM read operation is much less than that of DRAM, read-intensive variables are more likely to be allocated to the NVM. By comparing the energy consumption when each variable is located in the DRAM or NVM, the code profiler determines the variables which fit the NVM and stores the variable list on a file. The variable lists consist of a variable name, a line number of the declaration, and a file name. Table 1 lists the information acquired by the code profiler for global and heap memory objects.

With information provided by the code profiler, the code generator transforms the chosen memory objects in the program code with persistent object programming interfaces. The code generator adds persistent object APIs to original application code to efficiently utilize NVMs. Since the code conversion using HEAPO APIs can be a simple routine with simple string manipulation, we chose to

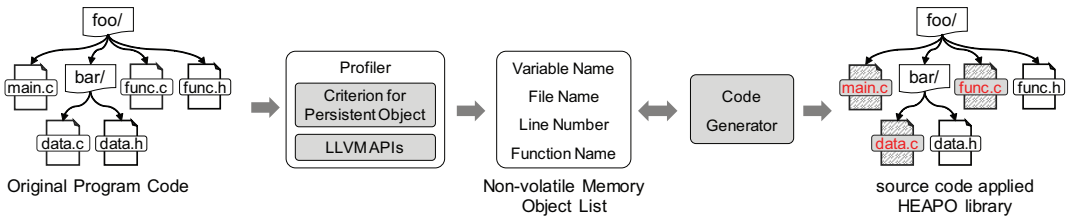

Figure 3. Code Profiling and Regenerating Process of Code Regenerator

use script language and the regular expression that is available on every Linux machine. Regular expression is a powerful tool that allows searching and substituting strings with rules defined in a formal language. Most text editors on UNIX-like systems provide string substitution, deletion, insertion, etc. that are based on regular expression. Using regular expression, the code generator first finds precise positions to be transformed with persistent object APIs in the original program code. The chosen strings in the original program code are then converted with proper NVM supporting interfaces. Fig. 3 shows the overall process of the code regeneration by the code profiler and the code generator. In Fig. 3, the code profiler first examines memory objects of the original source codes in the `foo` directory and stores the profiling result and NVM object list to a file. From the profiling result, the code generator then inserts additional codes to utilize HEAPO APIs to the original source codes (`main.c`, `func.c`, and `data.c`) including the object variables.

3.2 Code Generation for HEAPO persistent objects

To make use of NVM devices, persistent objects must be created in the persistent heap of HEAPO which is mapped to an NVM. It is important for the generator to make sure that the name of a persistent object is not used for other objects. In order to solve the problem, the code generator uses the name of the directory where original program code is contained as the default name of a persistent object. In the transformed code, all persistent object programming APIs use the name as a parameter.

After setting the name, HEAPO creates a new persistent object or maps an existing persistent object in the process address space before manipulating the object with persistent object programming APIs. As described in Section 2.1, `pos_create()` creates a new persistent object, and `pos_map()` maps an existing persistent object. Therefore, if `main()` is called when process runs, the code generator inserts function call syntax for `pos_create()` and `pos_map()` at the beginning of `main()`. Original sample code (Code 2) is shown as follows:

Code 2. Original main() function

```
7  ...
8  int main(int argc, void **argv){
9     int a;
10    ...
```

Code generator uses `if` statement to call `pos_create()` or `pos_map()`. If `pos_create()` fails, it means that a persistent object with the same name already exists, and therefore, `pos_map()` is called instead to use the object. The generated code consists of more than one statements in multiple code lines. We use a block to group new code together to prevent conflict or inconsistency with the original code. The transformed code from the original code is shown in Code 3:

Code 3. Regenerated main() function

```
7  ...
8  int main(int argc, void **argv){
9
10    {/* create or map persitent object */
11    if(pos_create("object_name") < 1){
12       if(pos_map("object_nmae") < 1){
13             return -1;
14          }
15       }
16    }
17    int a;
18    ...
```

The code profiler creates a CSV (comma separated values) file with the name of a file, the line number, and the variable name used in `malloc()` (Table 1) as a record. For variables that reside in heap area, HEAPO uses `pos_malloc()` to allocate a chunk of memory within the persistent heap. For the allocation process, the API requires the size of memory length and the name of a persistent object. The return value is a memory address pointer in the persistent heap. The code generator extracts the size from the original `malloc()` and retrieves the name of the object from `pos_create()` to generate `pos_malloc()` code. Since the persistent heap and the normal heap are managed independently in HEAPO managed process address space, memory objects allocated with `malloc()` and `pos_malloc()` are placed differently in the normal heap and the persistent heap, respectively (Fig. 1).

Memories allocated by `malloc()` should be deallocated by `free()` and memories allocated by `pos_malloc()` must be deallocated by `pos_free()`. A pointer to an object allocated with `malloc()` or `pos_malloc()` can be referenced by one or more pointer variables. Due to the problem, it is hard to know whether a pointer variable is for a memory object allocated with `pos_malloc()` or not. Therefore, we need to insert a code to find the location of the memory to distinguish between the memory from the original heap and the memory from persistent heap when the memory is deallocated. In the code generator, either `free()` or `pos_free()` can be chosen to free memory objects based on their location because the address of persistent heap in HEAPO with 64bit address space is defined from `0x5FFEF8000000` to `0x7FFEF8000000` as mentioned in Section 2.1. The extracted name from the code generator is used as the argument of `pos_free()`, and the newly created lines are grouped into a block similar to the memory allocation process. Memory deallocation example is shown in the following codes (Code 4 for the original code and Code 5 for the transformed code):

Code 4. Original memory free code

```
46  ...
47  if((p = L->Header) && (q == p)){
48     L->Header = P->link;
49     free(q);
50     P =  L->Header;
51     ...
```

Code 5. Regenerated memory free code

```
46  ...
47  if((p = L->Header) && (q == p)){
48     L->Header = P->link;
49     {if(q >= 0x5FFEF80000 && q < 0
           x7FFEF80000){
50           pos_free("object_name", q);
51        }else{
52           free(q);
53        }
54     }
55     P =  L->Header;
56     ...
```

A persistent variable is declared with the keyword named `__attribute__((section("POS")))` and allocated in POS segment during compile time like a normal global variable. The code generator transforms the global variables chosen by the code profiler to corresponding persistent variables using the keyword. For instance, a global variable in Code 6 is converted into a persistent variable by appending the keyword at the end of the variable name as in Code 7.

Code 6. Declaration of a global variable

```
20  ...
21  int array[10] = {0,};
22  ...
```

Code 7. Declaration of a persistent global variable

```
20  ...
21  int array[10] __attribute__ ((section("POS"
        )))= {0,};
22  ...
```

Note that the keyword in declaration syntax has an effect on other variables that are declared at the same line. If the keyword is located at the beginning of variable declaration syntax, all variables in the same line are declared as persistent variables. To declare a persistent variable, the keyword should be placed just after the variable name. Finally, when code conversion is done, the code generator inserts HEAPO library header file named `pos-lib.h` into all regenerated files to include HEAPO library API.

4 EXPERIMENT

To evaluate the effectiveness of code regeneration, we implement the code regenerator with four widely used applications such as `Mobibench` [14], `susan`, `bitcount`, and `stringserarch` from `Mibench` [9]. `Mobibench` is a benchmark tool designed for measuring IO performance of SQLite on Android file system. `susan` is an application that highlights corner and edge of an image file. `bitcount` is a program to count the number of bits in an array of integers. `stringsearch` is a benchmark which searches given words in phrases.

First, we profiled application codes to select memory objects that can be mapped to a persistent region and used the code regenerator to re-write application codes with persistent object programming APIs of HEAPO. Second, we compiled the generated application codes and executed the binary files on HEAPO implemented in Linux 2.6.32. Then, we verified that the regenerated applications show the same results to the original applications when they run on a conventional operating system. Finally, we compared the power consumption of the regenerated applications against the original applications. We implemented Code Regenerator in a desktop PC with Intel i5-2500, 8 GB DRAM, and 1 TB 7,200 RPM HDD. We set the DRAM read/write energy consumption to 0.7nJ and the STT-MRAM read/write energy consumption to 0.4nJ and 2.3nJ, respectively [24].

Table 2. Result of Code Regeneration (N: Objects allocated in NVM area, T: Total number of objects in the program)

Application	Global N/T	Heap N/T	Execution Time		Error
			Original	HEAPO	
Mobibench [14]	12(235)	1(15)	0.901s	1.647s	x
susan [9]	0(33)	3(9)	0.065s	0.157s	x
bitcount [9]	1(18)	0(1)	0.112s	0.218s	x
stringsearch [9]	1(233)	0(0)	0.002s	0.030s	x

4.1 Verification of Code Regeneration

Table 2 shows the number of memory objects that fit persistent object, an execution time of an original program, and an execution time of a corresponding regenerated program. In the case of `susan`, there were three out of nine heap objects that fit persistent variable. `bitcount` and `stringsearch` do not have any objects that can be mapped to persistent objects, but there is one global variable. Therefore, declaration syntax of the selected global variable of `bitcount` and `stringsearch` is regenerated. We found that Mobibench produces different results depending on the benchmark option. In this experiment, we set the mobibench option that generates the largest number of memory objects to convert.

Execution time in Table 2 represents execution times of the original program and a regenerated program, respectively. Though original and regenerated programs are executed with the same option, the execution time of a regenerated program increases by x1.8 to x15 times than that of the original program. This is because the regenerated program has to pass HEAPO software layer to manage persistent memory object when the program accesses and uses persistent memory objects. On the other hand, the original program that runs on a conventional operating system refers to page table to access physical memory where data segment is loaded. In all the test workloads, we confirmed that the regenerated applications generate the same results to the original applications, which means that the regenerated codes are implemented without any error ('Error' column in Table 2).

4.2 Energy Consumption

Fig. 4 and Table 3 show the energy consumption of the regenerated application compared with that of the original application. It shows the power consumption for when all variables are placed in DRAM, all variables are placed in NVM, and variables are allocated in both DRAM or NVM based on the code regenerator's selection (`Hybrid`), respectively. Based on the energy consumption results normalized to the baseline DRAM case, regenerated applications (`Hybrid`) show 22% ~ 44% energy savings compared to the original applications. In particular, the energy consumption is significantly reduced in `susan`

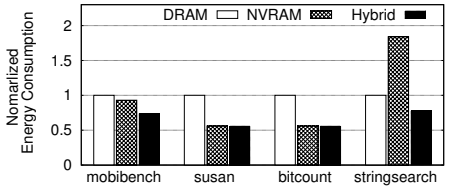

Figure 4. Normalized Energy Consumption

Table 3. Comparison of the Energy Consumption

	Mobibench	susan	bitcount	stringsearch
DRAM	651 mJ	53.9 J	25.1 J	516 mJ
NVM	608 mJ	30.2 J	14 J	950 mJ
Hybrid	479 mJ	30 J	14 J	402 mJ

and `bitcount`, where the number of memory reads is much higher than that of memory writes. In the `mobibench` and `susan` workloads, the total number of memory reads for the HEAP or Global variables is 6× and 647× higher than that of memory writes, respectively. In the `bitcount` workload, the memory reads for the global variables, `bits` and `bits1`, occurred 34,875,009 times without memory writes. On the other hands, In the `stringsearch`, the number of memory writes (349,005) is close to the number of memory reads (367,254). This results in an 84% increase in energy consumption compared to the original application if all variables are placed in NVM. From the results, we confirmed that the proper allocation of heap and global variables through the code regenerator is necessary to effectively reduce the energy consumption of an application.

5 CONCLUSION

In this paper, we propose a method to convert a program that runs on a conventional operating system into a program runs on a target NVM platform named HEAPO. We also implement a tool named a *Code Regenerator* that automates the code regeneration based on profiling. The proposed method for the code generation allows converting programs automatically without prior knowledge about NVM platform and its programming interfaces. Since there are several NVM software platforms support HEAPO-like persistent object programming APIs, the proposed method can be also applied in other NVM software platforms with a small modification. The regenerated applications reduce the energy consumption by up to 44% compared to the original applications.

6 ACKNOWLEDGMENTS

This research was supported by Basic Research Lab Program through the NRF funded by the Ministry of Science ICT&Future Planning(No. 2017R1A4A1015498), the BK21 plus program through the NRF funded by the Ministry of Education of Korea, the ICT R&D program of MSIP/IITP (R7117-16-0232, Development of extreme I/O storage technology for 32Gbps data services), the Ministry of Science ICT&Future Planning under the ITRC support program (IITP-2016-H8501-16-1006) supervised by the IITP, and the fund of research promotion program, Gyeongsang National University, 2017.

REFERENCES

[1] Raja Appuswamy, Matthaios Olma, and Anastasia Ailamaki. 2015. Scaling the Memory Power Wall With DRAM-Aware Data Management. In *Proc. of the 11th ACM International Workshop on Data Management on New Hardware (DAMON).*

[2] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2011. Operating System Implications of Fast, Cheap, Non- volatile Memory. In *Proc. of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS).*

[3] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. 2016. NVMOVE: Helping Programmers Move to Byte-Based Persistence. In *Proc. of the USENIX Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW).*

[4] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. *ACM SIGARCH Computer Architecture News* 39, 1 (March 2011), 105–118.

[5] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. 2016. Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials* 18, 1 (2016), 732–794.

[6] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2016. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proc. of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC).*

[7] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proc. of the 9th ACM European Conference on Computer Systems (Eurosys).*

[8] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software Persistent Memory. In *Proc. of the USENIX Conference on Annual Technical Conference (ATC).*

[9] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the 4th IEEE International Workshop on Workload Characterization (WWC-4).*

[10] Thomas Haywood-Dadzie. 2015. Hybrid memory profiler. (2015). https://github.com/sdmlab/NVRAMPF.

[11] Yiming Huai. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *Association of Asia Pacific Physical Societies (AAPPS) Bulletin* 18, 6 (2008), 33–40.

[12] Taeho Hwang, Jaemin Jung, and Youjip Won. 2014. HEAPO: Heap-Based Persistent Object Store. *ACM Transactions on Storage (TOS)* 11, 1, Article 3 (Dec. 2014), 21 pages.

[13] S James, P Arujo, and A Carlos. 1989. Ferroelectric memories. *Science* 246, 4936 (1989), 1400–1405.

[14] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. 2013. Framework for analyzing android i/o stack behavior: from generating the workload to analyzing the trace. *Future Internet* 5, 4 (2013), 591–610. http://www.mobibench.co.kr/

[15] Jaemin Jung, Youjip Won, Eunki Kim, Hyungjong Shin, and Byeonggil Jeon. 2010. FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage. *ACM Transactions on Storage (TOS)* 6, 1, Article 3 (April 2010), 25 pages.

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transfor- mation. In *Proc. of the IEEE International Symposium on Code Generation and Optimization (CGO).*

[17] Cheolhee Lee, Taeho Hwang, and Youjip Won. 2014. Improving I/O Performance in Smart TVs. In *Proc. of the IEEE Future Internet of Things and Cloud (FiCloud).*

[18] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *Proc. of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage).*

[19] Sparsh Mittal and Jeffrey Vetter. 2016. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (May 2016), 1537–1550.

[20] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase- change Memory Technology. In *Proc. of the International Symposium on Computer Architecture (ISCA).*

[21] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for Non-Volatile Main Memory Is Hard. In *Proc. of the 9th ACM Asia-Pacific Workshop on Systems (APSys).*

[22] John Rosenberg, Alan Dearle, David Hulse, Anders Lindström, and Stephen Norris. [n. d.]. Operating System Support for Persistent and Recoverable Computations. *Commun. ACM* ([n. d.]).

[23] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *ACM SIGPLAN Notices* 47, 4 (March 2011), 91–104.

[24] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. 2009. Hybrid Cache Architecture with Disparate Memory Technologies. In *Proc. of the International Symposium on Computer Architecture (ISCA).*

[25] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *Proc. of the ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC).*