Program Code Regeneration Method for Non-volatile Memory Platform

Seongsu Lee Youjip Won

Hanyang University, Seoul, Korea
{su880214 | yjwon}@hanyang.ac.kr

Abstract: Non-volatile memory is a promising material that covers both cache and secondary storage in a memory hierarchy. Many software platforms exploiting non-volatile have been developed and provide user-level programming interfaces. However, common programs cannot allocate non-volatile memory. In this paper, we introduce a code generation method for HEAPO that is one of non-volatile memory software platforms. With the method, a program designed for a legacy operating system using DRAM as main memory is converted to a program running on the non-volatile memory platform without code modification manually.

Keywords: Non-volatile memory; Code generation;

1 Introduction

DRAM is used as main memory in almost computer systems from a tiny embedded system to a massive database server. DRAM has critical limits such as power consumption, scaling capacity, data recovery overhead and serialization overhead. Non-volatile memory devices have been developed, and each device has different characteristics. So, they are expected to be used for each component in the memory hierarchy.

While non-volatile memory devices have been developed, many researches in software have been attempted on filesystem [1-3] and persistent object programming interfaces [4-7]. Persistent object means that data has to be kept permanently and can be accessed by processes or an operating system until it is removed [8]. Software platforms for a persistent object are developed to benefit a persistent memory object instead of a persistent object in a secondary storage. However, most programs are designed to run on a legacy operating system that does not provide non-volatile memory.

To make a program use non-volatile memory, source code of the program can be modified to use persistent object programming libraries which are provided by non-volatile memory platforms. Each non-volatile memory platform has its own programming model and APIs. Thus, a user has to learn each platform programming model and the way on using APIs.

In this paper, we introduce an automation technique to convert a common program to a program that runs non-volatile memory platform without manual code modification. With the proposed method, a user does not need to know how to apply programming models and APIs to existing programs.

2 Background

In this section, we explain HEAPO [4] that is one of non-volatile memory software platform. Fig. 1 shows a virtual address space in HEAPO. A new segment named persistent heap is defined in the address space, and persistent objects are allocated in the persistent heap. Pages in the persistent heap are mapped to physical page frame from non-volatile memory. HEAPO is designed for a hybrid system that uses STT-MRAM [9] and DRAM as main memory.

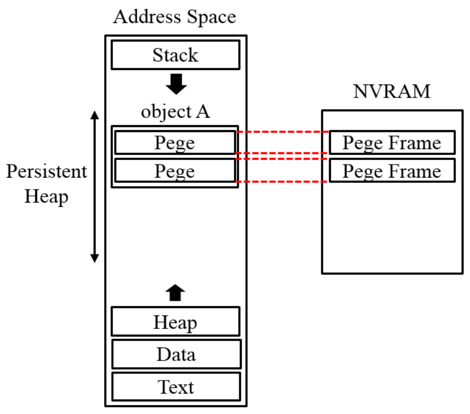


Figure 1. Address space in HEAPO and NVRAM

HEAPO has its own namespace like a filesystem. A process uses a persistent object based on its name like a file. HEAPO provides programming library and APIs to manipulate persistent objects for a programmer. If a programmer uses HEAPO programming library on coding a program, the process allocates non-volatile memory to store a persistent object. The allocated persistent objects are kept permanently in NVRAM, so that can be used by processes or system afterward.

The programming model of HEAPO is simple. HEAPO provides C Programming language user-level library. The most necessary APIs are pos_create(), pos_map(), pos_delete(), pos_malloc() and pos_free(). First, a process creates a persistent object by pos_create() with a name that does not have to be duplicated with other persistent objects' names. If a persistent object that has the same name exists, a process fails to create a persistent object with the name. In the above-mentioned case, the process maps a persistent object existing already in NVRAM to its process address space by pos_map(). Since a persistent object is created or mapped, the process can allocate dynamically a memory chunk in bytes from a persistent object by pos_malloc(). The way on using pos_malloc() is almost similar to malloc() from C standard library. As expected by the names of APIs, pos_delete() is used to delete a persistent object in the persistent heap, and pos_free() is the same role with free() from C standard library.

3 Code regeneration method based on profiling

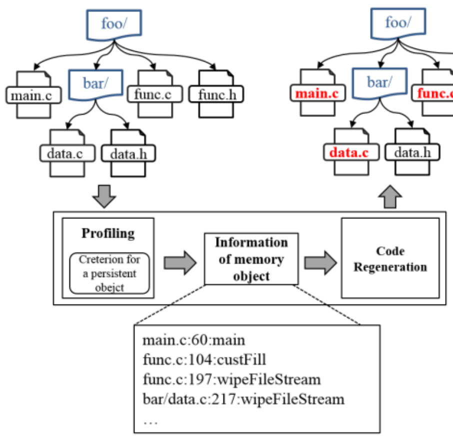

foo/
bar/
main.c
func.c
func.h
data.c
data.h
Profiling
Creterion for a persistent obejct
Information of memory object
Code Regeneration
main.c:60:main
func.c:104:custFill
func.c:197:wipeFileStream
bar/data.c:217:wipeFileStream
...

Figure 2. Process of code regeneration

In this section, we describe the automation technique to convert a program to a program that runs non-volatile memory platform. We consider memory objects in heap segment to apply the non-volatile programming model and APIs. In the process address space, memory objects are stored in data (bss), heap, and stack segment. Memory objects in each segment has its own properties and purposes in the legacy programming model. A memory object in heap is dynamically allocated while the program is running which is generally used to build a data structure that are dynamically updated. Also, memory objects in heap can be accessed throughout the program by referencing the memory address unless they are deallocated. On the other hand, a local variable that is given a local scope can be only accessed within in a function and a block where the variables are declared in. Local variables are automatically allocated and freed. In the legacy programming model, recursive function and multi-threaded programming exist, and a local variable is used throughout the models. And a global variable in data (bss) segment including local variable declared with static keyword has global scope meaning that it is visible throughout the program. However, in the proposed methods, programming APIs are used. To convert global variables, modifying loader may be required. For the reasons, we rule out local variables and global variables as consideration of persistent object. Fig 2 shows the code regeneration process.

3.1 Profiling program code

The procedure of converting a program consists of two components. First of all, a program code is profiled to find memory objects among all objects in heap that will be changed to a persistent object. And then, the APIs of HEAPO are applied to the program code based on information that are gathered by profiling. Fig 2 shows the code regeneration process.

Code profiler uses front-end library of low-level virtual machine (LLVM) [10] compiler framework. The front-end library provides programming APIs to modify LLVM IR code for a programmer. By the APIs, a programmer can modify intermediate representation (IR) code of a program. Code Profiler inserts instructions of a function call back and forth on instructions of memory read/write to count memory accesses.

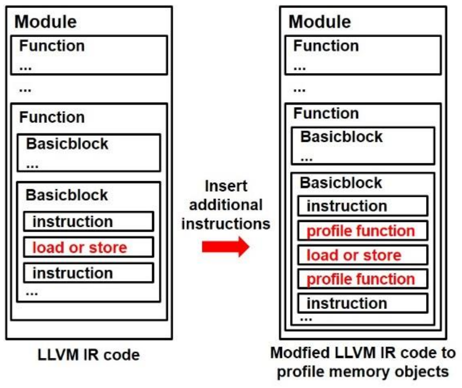


Figure 3. Modification on LLVM IR code by code profiler

Fig 3 represent modifying LLVM IR code to gather information of memory accesses. Gathered information is saved in a comma-separated values (CSV) [11] format in a file with added IR code when the process terminates. A program through profiling executes more instruction to gather memory information and take longer execution time than the execution time of the original programs. Code profiler uses dynamic profiling approach, so the result of profiling and generated code can adapt to change the memory access patterns of the application.

3.2 Criterion for a persistent object

In order to apply HEAPO programming library to a program code, we should find a proper memory object as a persistent object. While a process is running, the process accesses memory to read and write frequently. Each memory device has different characteristics. By code profiling, the read and write counts of each memory object are gathered. And then, energy consumption of memory objects allocated on DRAM and STT-MRAM is calculated.

Table I Memory parameter (45nm) [12]

RAM	Latency (cycles)	Energy (nJ)
DRAM	24	0.72
STT-MRAM	read: 20 write: 60	read: 0.4 write: 2.3

For the energy consumption for read and write memory, we use the parameter in Table I [12] in which STT-MRAM requires lower read energy consumption while it consumes more energy to write than DRAM. The read and write energy consumption for DRAM is 0.7nJ for both, and STT-MRAM is 0.4nJ and 2.3nJ for read and write, respectively. The calculated energy consumption of DRAM and STT-MRAM are compared. Then, memory objects that consume lower energy with STT-MRAM are chosen as persistent objects.

3.3 Applying persistent object APIs

To use a persistent object, a process must create a persistent object before manipulating a persistent object. Like a file, a persistent object has its own unique name. Thus, a name of a persistent object must not be overlapped with other objects. The name of directory where original program code resides is used as a default name of a persistent object. In generated code, all persistent object programming APIs use the name as a parameter. After setting the name, a new persistent object is created or a persistent object existing already is mapped into process address space to use. A process has to create or map a persistent object to call other persistent object programming APIs to manipulate. pos_create() creates a new persistent object, and pos_map() maps a persistent object that is already created. First of all main() is called when process runs, so function call syntax for pos_create() and pos_map() is inserted at the beginning of main(). Then, the API that creates a persistent object is generated at the beginning of main() of the program.

```
#include <stdio.h>
#include "pos-lib.h" /* HEAPO library header */
#define BUF_SIZE 1024

int main(int argc, void **argv){
  { /* create or map persitent object */
    if(pos_create("object_name") < 1){
      if(pos_map("object_nmae") < 1){
        return -1;
      }
    }
  }
  int *tmp;
  ...
  /* allocate a memory chunk in bytes */
  tmp = (char *)pos_malloc("object_name", reclen *
      2);

  {/* freeing memory according to address */
    if(stmp >= 0x5FFEF80000 && stmp < 0x7FFEF80000){
        pos_free("object_name", stmp);
    }else{
        free(stmp);
    }
  }
  ...
```

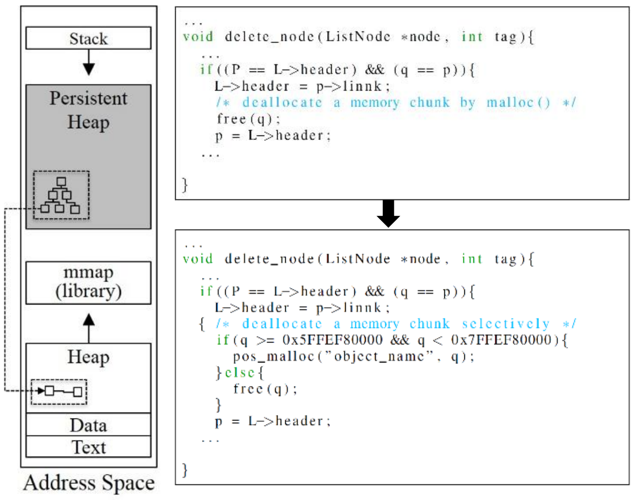
Figure 4. Programming model of HEAPO and APIs

Profiler saves information of memory object allocation code as a text file. The information includes a name of a file, a line number, and a variable name of malloc(). pos_malloc() allocates a chunk of memory within persistent object created by pos_create(). The API takes a size of memory length and additionally a name of a persistent object. The return value is a memory address pointer. The name from pos_create() and the size from malloc() in original code are used as parameters for pos_malloc(). The size from malloc() is extracted using regular expression. Fig 4 show a simple code of the HEAPO programming model and how to apply the APIs to a program code.

3.4 Selectively deallocate memory

A program code through code regeneration includes both memory allocation functions for DRAM and NVRAM. In a process address space, the persistent heap is separated from the heap. Memory chunks allocated by malloc() should be deallocated by free(), and memory chunks allocated by pos_malloc() also must be deallocated by pos_free(). Generally, dynamic memory allocation is used to make a data structure like a linked list. The problems are that malloc() and pos_malloc() together exist in generated code based on profiling and a pointer by malloc() or pos_malloc() can be referenced by several pointer variables. Deallocating only the pointer variable that is in the information by profiling code may occur critical errors. Thus, verifying memory location is required to check whether the memory is from the heap or the persistent heap when memory is freed in generated code.

Persistent heap of HEAPO in 64bits address space starts at 0x5FFEF8000000 and ends at 0x7FFEF8000000. Memory free syntax should be generated with a conditional statement. The generated conditional statement is based on the memory addresses to choose free() and pos_free(). When generating code for deallocating memory, a name of pointer variable that is used for an argument of free() are extracted by regular expression. Generated memory deallocation syntax consists of more than one statement and is grouped into a block. Since the generated code spans multiple lines, we use a block to group new code together to prevent conflict with the original code. Fig 5 show how to regenerate deallocating memory code.



```
...
void delete_node(ListNode *node, int tag){
  ...
  if((P == L->header) && (q == p)){
    L->header = p->linnk;
    /* deallocate a memory chunk by malloc() */
    free(q);
    p = L->header;
  ...
}
```

```
...
void delete_node(ListNode *node, int tag){
  ...
  if((P == L->header) && (q == p)){
    L->header = p->linnk;
  { /* deallocate a memory chunk selectively */
    if(q >= 0x5FFEF80000 && q < 0x7FFEF80000){
      pos_malloc("object_name", q);
    }else{
      free(q);
    }
    p = L->header;
  ...
}
```

Figure 5. Selectively deallocating memory

4 Experiment

To demonstrate the proposed method, we use applications named mobibench [13] and susan from mibench [14]. We verify that a program by the technique run HEAPO and the output of the program is same as the out from the original program. Table II shows a number of heap memory objects that fit NVRAM, an execution time of an original program and an execution time of a program coded with HEAPO programming library.

Table II Code regeneration result

Program	Memory object	Execution Time	
		original	HEAPO
mobibench [13]	2(15)	0.901s	1.647s
susan [14]	3(9)	0.065s	0.157s

Mobibench is a benchmark tool for simulating IO characteristics. In the result, it has 2 memory objects among 15 objects in the heap that fits NVRAM. The two memory objects are allocated in NVRAM by HEAPO programming library. The execution time of a program that is regenerated by the proposed method is increased by 1.8x because HEAPO has its own software layer to manage a persistent object in persistent heap. The original program uses malloc() from C standard library. On the other hand, the regenerated program uses HEAPO library, and the process has to pass the HEAPO software layer to manipulate a persistent object. The overhead comes from more changing mode from user to kernel and searching a persistent object in the namespace of HEAPO. Susan is a program that highlights edges of an image. In the case of this program, 3 objects among 9 objects fit NVRAM. Also, the execution time is increased by 2.4x than the execution time of the original program.

The result shows that programs have even a small number of persistent objects and regenerated program code by the proposed method runs without manual code modification. However, the longer execution time can be critical issues, so the proposed method cannot match some program.

5 Conclusions

In this paper, we introduce a technique that converts a program to a program running on NVRAM software platform without manual code modification. To automate generating code, we propose the criterion for a persistent object using energy consumption. The technique allows converting programs without learning NVRAM software platform and the way on using its APIs. Including HEAPO, many NVRAM software platforms have been developed. The technique can be applied to other NVRAM software platforms. As a future work, we try to the technique to other NVRAM software platform to provide an environment to make many various programs run on NVRAM software platforms.

Acknowledgements

This work was supported by the BK21 plus program through the National Research Foundation (NRF) funded by the Ministry of Education of Korea, Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) [R7117-16-0232, Development of extreme I/O storage technology for 32Gbps data services], and the ICT R&D program of MSIP/IITP [R0601-16-1063, Software Platform for ICT Equipments].

References

[1] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon, “Frash: Exploiting storage class memory in hybrid file system for hierarchical storage,” ACM Transactions on Storage (TOS), vol. 6, no. 1, p. 3, 2010.

[2] X. Wu and A. Reddy, “Scmfs: a file system for storage class memory,” in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011, p. 39.

[3] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, “System software for persistent memory,” in Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014, p. 15.

[4] T. Hwang, J. Jung, and Y. Won, “Heapo: Heap-based persistent object store,” ACM Transactions on Storage (TOS), vol. 11, no. 1, p. 3, 2015.

[5] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” ACM SIGPLAN Notices, vol. 46, no. 3, pp. 91– 104, 2011

[6] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, “Software persistent memory,” in Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), 2012, pp. 319–331.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories,” in ACM SIGARCH Computer Architecture News, vol. 39, no. 1. ACM, 2011, pp. 105–118.

[8] J. Rosenberg, A. Dearle, D. Hulse, A. Lindstr¨om, and S. Norris, “Operating system support for persistent and recoverable computations,” Communications of the ACM, vol. 39, no. 9, pp. 62–69, 1996

[9] Y. Huai, “Spin-transfer torque mram (stt-mram): Challenges and prospects,” AAPPS Bulletin, vol. 18, no. 6, pp. 33–40, 2008.

[10] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004, pp.75–86.

[11] Y. Shafranovich, “Common format and mime type for commaseparated values (csv) files,” RFC, 2005.

[12] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, “Hybrid cache architecture with disparate memory technologies,” in ACM SIGARCH computer architecture news, vol. 37, no. 3. ACM, 2009, pp. 34–45.

[13] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won, “Framework for analyzing android i/o stack behavior: from generating the workload to analyzing the trace,” Future Internet, vol. 5, no. 4,

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” in Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on. IEEE, 2001, pp. 3–14.