Using File Systems for Non-volatile Main Memory Management

Shuichi Oikawa
Faculty of Engineering, Information and Systems, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, Japan

Keywords: Operating Systems, Memory Management, Non-volatile Memory.

Abstract: Non-volatile (NV) memory is next generation memory. It provides fast access speed comparable to DRAM and also persistently stores data without power supply. These features enable NV memory to be used as both main memory and secondary storage. While the active researches have been conducted on its use for *either* main memory or secondary storage, they were conducted independently. This paper proposes the integrated memory management methods, by which NV memory can be used as *both* main memory and secondary storage. The proposed methods use file systems as their basis for NV memory management. Such integration enables the memory allocation for processes and files from the same source, and processes can take advantage of a large amount of physical memory used for both main memory and storage. We implemented the proposed memory management methods in the Linux kernel. The evaluation results performed on a system emulator show that the memory allocation costs of the proposed methods are comparable to that of the existing DRAM and are significantly better than those of the page swapping.

1 INTRODUCTION

Non-volatile (NV) memory is next generation memory. It provides fast access speed comparable to DRAM and also persistently stores data without power supply. These features enable NV memory to be used as main memory and secondary storage. Especially, the active researches have been conducted to utilize phase change memory (PCM) as main memory from the computer architecture's point of view (Lee, et. al., 2009; Qureshi, et. al., 2009; Zhou, et. al., 2009; Qureshi, et. al., 2010). Since these were the researches on the computer architecture, the operating system (OS) takes only a minor role (Zhang and Li, 2009; Mogul, et. al., 2009). There are also the researches that construct file systems on NV memory by taking advantage of its byte addressability (Condit, et. al., 2009; Wu and Reddy, 2011).

While these researches on the use of NV memory for either main memory or storage have been performed as described above, they were conducted independently. Since NV memory can be used for both main memory and storage, their management can be integrated. Such integration enables the memory allocation for processes and files from the same source, and processes can take advantage of a large amount of physical memory used for both main memory and storage. Therefore, integrating memory management with a file system on a NV main memory system enables the improvement of system performance by removing the paging swapping between main memory and storage. While the advantage brought by the integration were discussed (Bailey, et. al., 2011; Jung and Cho, 2011), there has been no research effort to realize it in an actual OS.

This paper proposes the integrated memory management methods, by which NV memory can be used as both main memory and storage. The two, direct and indirect, methods that use file systems as their basis for NV memory management are described. The direct method *directly* utilizes the free blocks of a file system by manipulating its management data. The indirect method *indirectly* allocates blocks through a file that was created in advance for the use of main memory. These methods have their own advantages and disadvantages; thus, each method meets different requirements.

We implemented these memory management methods in the Linux kernel. When the kernel tries to allocate a physical memory page from the NV memory, the kernel directly or indirectly consults the file system and takes a free block from it. By making the block size of the file system the same as the page size and using the XIP (eXecution In Place) feature, allocated file system blocks can be used as physical memory pages and be directly mapped into a virtual memory address space without copying. The evaluation results performed on a system emulator show that the memory allocation costs of the proposed methods are comparable to that of the existing DRAM and are significantly better than those of the page swapping. To the best of our knowledge, we are among the first to design and implement the integration of main memory and storage.

This paper focuses on the integration of the main memory and file system management; thus, the evaluation results presented in Section 4 do not take into account the difference of access latencies and treat DRAM and NV memory the same. This is because NV memory technologies are still under active development and it is possible that some of them will perform comparably to DRAM. For example, the performance of STT-RAM is comparable to DRAM, and there is no need to treat it differently (Park, 2012).

The rest of this paper is organized as follows. Section 2 describes the related work. Section 3 presents the design and implementation. Section 4 describes the current status and shows experiment results. Finally, Section 5 concludes this paper.

2 RELATED WORK

There are only a few papers that describe the integration of main memory and storage. (Bailey, et. al., 2011) discusses various possibilities, including the integration of main memory and storage, made possible by employing NV memory as main memory. (Jung and Cho, 2011) describes the policy and possible effect of the integration. Neither of them, however, realized the integration. This paper describes the methods to realize it and presents their design and implementation in the Linux kernel.

There are a number of researches conducted to enable byte addressable NV memory to be used as main memory (Lee, et. al., 2009; Qureshi, et. al., 2009; Zhou, et. al., 2009; Qureshi, et. al., 2010; Zhang and Li, 2009; Mogul, et. al., 2009). Since these are the researches of the computer architecture to integrate NV memory into main memory by overcoming its limitations, there is no consideration to integrate main memory and storage. On the other side, there are the researches that construct file systems on NV memory by taking advantage of its byte addressability (Condit, et. al., 2009; Wu and Reddy, 2011). Although these researches utilize the feature that enables NV memory to be used as main memory, they do not consider NV memory as main memory at all.

FlashVM (Saxena and Swift, 2010) and SSDAlloc (Badam and Pai, 2011) propose the methods to make usable memory spaces virtually larger by utilizing SSDs and making page swapping faster than the existing mechanism. While these improve the virtual memory system of the OS kernel, main memory and storage remain separated.

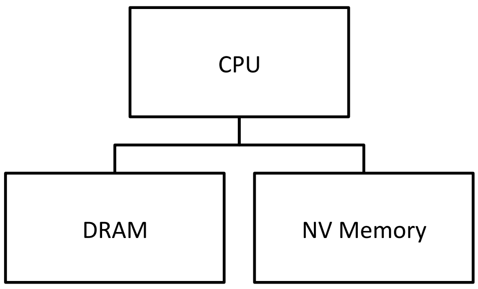

Figure 1: Target system memory structure.

3 DESIGN AND IMPLEMENTATION

This section describes the design and implementation that enable the integration of the main memory and storage management by employing byte addressable NV memory.

3.1 Target System Structure

Since there is no publicly available system that employs byte addressable NV memory as its main memory, we need to construct a reasonable target system structure. In this paper, we assume that 1) DRAM and byte addressable NV memory constitute the main memory of a system, and 2) DRAM and byte addressable NV memory are placed in the same physical address space. DRAM and byte addressable NV memory are connected to the memory bus(ses) of CPUs, and they are mapped in the same physical address space; thus, they can be accessed in the same way with appropriate physical addresses, and there is no distinction between them from the OS kernel. Figure 1 depicts the memory architecture of the target system.

Since our goal is to integrate the main memory and storage management, byte addressable NV memory takes the roles of both main memory and storage. Our methods construct a file system on NV memory. The file system is used to store directories and files, and these contents persist across the termination and rebooting of the OS kernel. The free blocks of the file system are used for main memory while a system is running. They are returned to the file system when a system terminates.

We consider this structure is reasonable when our primary target systems are clients devices, such as note PCs, tablets, and smart phones. Since these client devices do not require a large amount of stor-

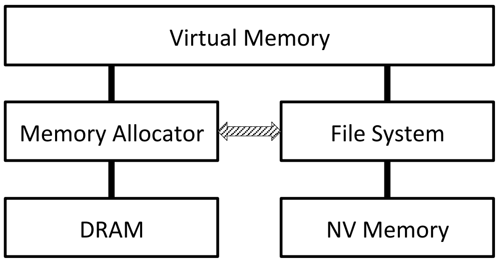

Figure 2: Virtual memory system architecture and its relationships with the memory allocator and a file system.

age spaces, NV memory suffices the needs of storage. While NV memory can constitute the whole memory of a system, DRAM is also useful to hold data regions, which are known to be volatile and overwritten soon. Examples of such regions include stacks and buffers used for data transfer. Thus, it should be reasonable to consider that a target system's memory consists of both DRAM and NV memory.

We employ the QEMU system emulator to construct the target system described above and to execute Linux on it. The more details of the emulation environment is described in Section 4.1.

3.2 Virtual Memory System

This section describes the virtual memory system architecture and its relationships with the memory allocator and a file system. Figure 2 depicts the architecture. The virtual memory system of the Linux kernel is built upon the memory allocator and file systems.

Traditionally, the memory allocator manages DRAM, and file systems manage storage. The virtual memory system uses the memory allocator to allocate physical memory pages from DRAM. It then maps the allocated DRAM pages in virtual address spaces.

The virtual memory system can also use XIP[1]-enabled file systems, such as Ext2 and PRAMFS, to directly map files in user process address spaces. Since the physical memory pages of the files are mapped through the virtual memory system, no copying of pages occurs between DRAM and NV memory. Since Ext2 and PRAMFS are the only XIP-enabled file systems that support both read and write, these are our target file systems to implement the integrated memory management methods described in the next section.

3.3 Integrating Main Memory and File System Management

This section describes the integrated memory management methods, by which NV memory can be used for the memory allocation for both processes and files. In order to use NV memory as main memory, physical memory pages need to be allocated from a file system. Therefore, in Figure 2, the double arrowed line that connects the memory allocator and a file system is a missing link. Making the memory allocator allocate physical memory pages from a file system connects the link; thus, it enables the integration of the main memory and storage management.

This paper proposes the two, direct and indirect, methods to connect the link. Both of them use file systems as their basis for NV memory management. The direct method directly utilizes the free blocks of a file system by manipulating its management data. The indirect method indirectly allocates blocks through a file that was created in advance for the use of main memory. These methods have their own advantages and disadvantages; thus, each method meets different requirements. The details of these methods are described below.

3.3.1 Direct Method

The *direct* method allocates free blocks from a file system for the use of main memory just as those are allocated for files. The allocation is done by finding free blocks and marking them allocated. Such information is stored in the management data of a file system; thus, this method requires the *direct* manipulation of the management data, and the additional code for the allocation and freeing needs to be implemented.

The advantage of the direct method is the tight integration of main memory and file system management. Any of the free blocks of a file system can be used for both main memory and files. The use of the free blocks is not decided and they remain free until their allocation; thus, this method does not waste the free blocks. The disadvantages are the dependency on the implementation of a file system and the crash recovery. The dependency issue involves two aspects, the effect of the internal structure to the performance and the implementation cost of the additional code. The crash recovery is required because the allocated blocks for the use of main memory do not belong to any file; thus, these blocks cause the inconsistency of a file system when crashed.

3.3.2 Indirect Method

The *indirect* method utilizes the blocks of a file that was created in advance to be used for main memory. This method does not need to consult the management data of a file system, but *indirectly* uses the blocks that

[1]XIP stands for eXecution In Place.

were allocated for a file. The content of the file is initialized by creating a linked list of its blocks; thus, all of the file's blocks need to be allocated when created. The allocation for main memory is done by simply taking blocks out of the linked list.

The advantages of the indirect method are contrary to the direct method. The indirect method does not depend on the implementation of a file system. The provision of the XIP feature of a file system requires the implementation of `get_xip_mem()` interface. The interface converts a file offset to its block address. By using this interface, the initialization of a file used for main memory can be done independently from the internal implementation of a file system. The linked list of the free blocks also makes the allocation cost independent from the internal implementation. Moreover, since the blocks used for main memory are allocated for a file, they do not cause the inconsistency of a file system when a system crashes. The disadvantage is that the blocks of a file used for main memory are preallocated; thus, their use is fixed for main memory. It is desirable to adjust the size of the preallocated file. While it is easy to extend the file size as more blocks need to be allocated for main memory, more work is necessary to shrink it.

4 EXPERIMENT RESULTS

This section first describes the evaluation method and then shows the experiment result that measures the allocation costs of main memory from file systems.

4.1 Evaluation Method

We employ the QEMU system emulator to construct the target system described in Section 3.1. The version of QEMU used for the evaluation is 1.0.1, and QEMU emulates x86_64. QEMU was modified to emulate NV memory that persists its contents across the termination and rebooting of the emulator. A file is used for the persistence of the NV memory. The file is mapped into the physical address space emulated by QEMU. The experiments described in this section were performed on QEMU invoked with the following options:

```
% qemu -m 128 -nvmemory \
  file=nvmemory.img,physaddr=0x100000000
```

With the above options, QEMU is invoked with 128MB DRAM along with the NV memory mapped from 0x100000000 of the physical address. The size of the file emulating NV memory (`nvmemory.img`) is 4GB. While the size of DRAM is passed to the Linux kernel through BIOS, the information of NV memory is not passed in order to make their management separate.

The modified QEMU executes the Linux kernel that includes the modifications of the proposed methods. The version of the Linux kernel modified and used for the experiments is 3.4.

The evaluation of execution costs needs to measure execution times. Times counted by the interrupts from a timer device are not accurate enough on system emulators. Instead, the number of instructions executed is used as the measure of execution costs. The `-icount 0` option of QEMU lets the TSC (time stamp counter) register count the number of instructions executed. The RDTSC instruction reads the value of TSC.

As described in Section 1, the following evaluation results do not take into account the difference of access latencies and treat DRAM and NV memory the same.

4.2 Page Allocation Costs

This section shows the measurement results of the allocation costs of main memory from file systems and compares them with those that use the page swapping mechanism. Integrating the main memory and file system management on NV memory brings a large amount of physical memory to processes; thus, such integration should be able to remove the necessity of the paging swapping and to improve the performance to allocate a large amount of memory. In order to verify the effectiveness of the integration, we executed a program that allocates a memory region by using `malloc()` and performs writes to the beginning of the page boundaries for the actual allocation of physical memory pages.

The measurements were performed for several cases, and Figure 3 shows the results. The figure compares the eight cases. DRAM (w/o swap) and DRAM (w/ swap) show the costs when only DRAM is used as main memory without and with a swap device, respectively. In these cases, a swap device for DRAM is a ramdisk created on NV memory. PRAMFS NV mem (direct) and PRAMFS NV mem (indirect) show the costs when PRAMFS is constructed on NV memory and the direct and indirect methods are used for main memory allocation from the file system, respectively. PRAMFS swap file shows the cost when only DRAM is used as main memory and a swap file created on PRAMFS is used as a swap device. In the cases of Ext2 NV mem (direct), Ext2 NV mem (indirect), and Ext2 swap file, Ext2 was constructed on NV memory.

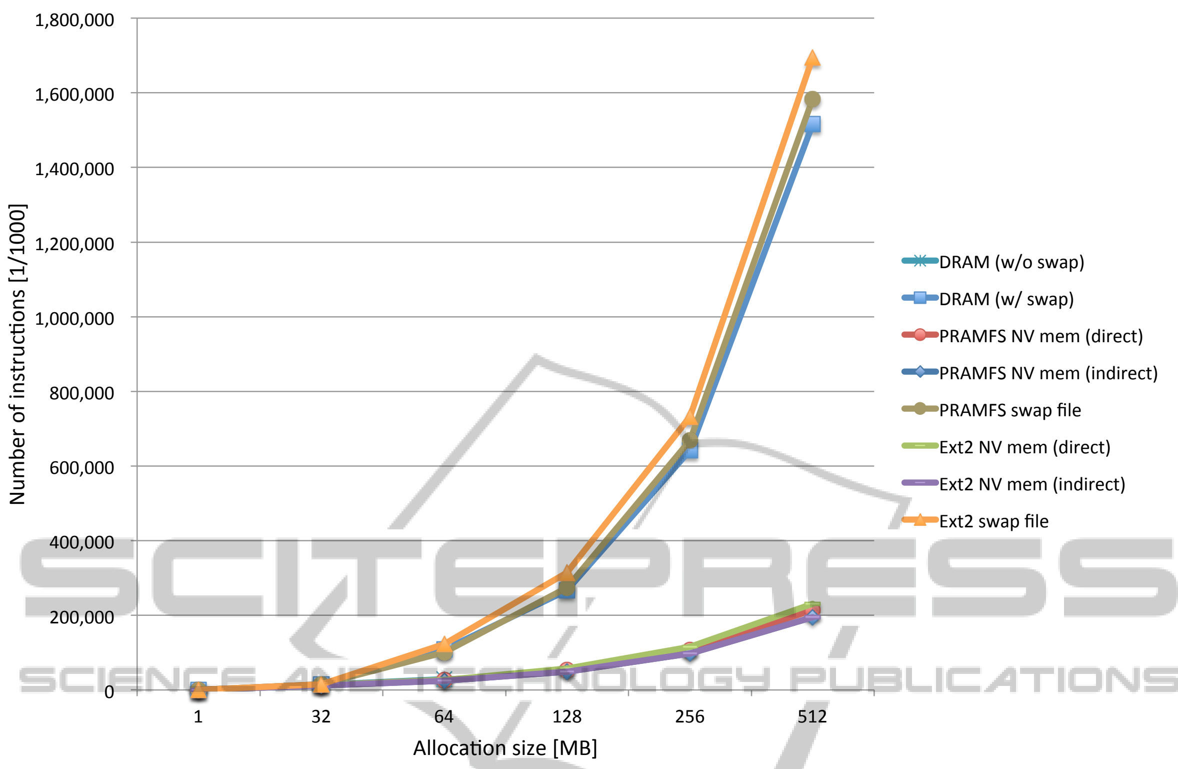

Figure 3: Comparison of page allocation costs.

The measurement results apparently show the efficiency of the memory allocation from NV memory and the inefficiency of the cases that use the page swapping. The cases that use the indirect method performs the best, and their results are basically the same because of their implementation independent from the underlying file systems. PRAMFS NV mem (direct) comes next to the indirect method with 10% overhead for the allocation of 512MB of memory. Ext2 NV mem (direct) poses 18% overhead. DRAM (w/ swap), PRAMFS swap file, and Ext2 swap file are approximately 7.8, 8.1, and 8.7 times as much as the indirect method, respectively.

For the both cases that use the direct method and the page swapping, Ext2 performs slower than PRAMFS, and Ext2 poses approximately 7% overhead over PRAMFS. Since these cases depend on the internal implementations of the file systems, the complicated mechanisms of Ext2, which have been developed and optimized for hard disk drives (HDDs), had a negative impact for the use for NV memory. Therefore, the internal implementations of the file systems affect the performance of the integrated management of main memory and a file system.

5 SUMMARY AND FUTURE WORK

Non-volatile (NV) memory is next generation memory. It achieves performance comparable to DRAM and also persistently stores data without power supply. These features enable NV memory to be used as main memory and secondary storage. While the active researches have been conducted on its use for either main memory or secondary storage, theses researches were conducted independently. This paper proposed the integrated memory management methods, by which NV memory can be used as both main memory and storage. The two methods that use file systems as their basis for NV memory management were described. The direct method utilizes the free blocks of a file system by manipulating its management data. The indirect method allocates blocks through a file that was created in advance to be used for main memory. We implemented these memory management methods in the Linux kernel. The evaluation results performed on a system emulator showed that the memory allocation costs of the proposed methods are comparable to that for the existing DRAM and are significantly better than those of the page swapping. To the best of our knowledge, we are among the first to design and implement it.

The integration of the main memory and file system management creates new possibilities of the usage of NV memory. This paper is just the starting point of various further investigations. We will investigate new possibilities of the use of NV memory.

REFERENCES

A. Badam and V. S. Pai. SSDAlloc: hybrid SSD/RAM memory management made easy. In *Proc. of 8th USENIX conference on Networked systems design and implementation (NSDI '11)*, 2011.

K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proc. of the 13th USENIX conference on Hot topics in operating systems (HotOS 13)*, 2011.

J. Condit, E. B. Nightingale, C. Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*, pp. 133-146, 2009.

J-Y. Jung and S. Cho. Dynamic co-management of persistent RAM main memory and storage resources. In *Proc. of the 8th ACM International Conference on Computing Frontiers (CF '11)*, 2011.

B. C. Lee, B. C. E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*, pp. 2-13, 2009.

J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proc. of the 12th conference on Hot topics in operating systems (HotOS '09)*, 2009.

Sung Wook Park: Overcoming the Scaling Problem for NAND Flash. *Flash Memory Summit*, 2012.

Protected and Persistent RAM Filesystem. http://pramfs.sourceforge.net/, 2012.

M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proc. of the 36th annual international symposium on Computer architecture (ISCA '09)*, pp. 24-33, 2009.

Qureshi, M.K.; Franceschini, M.M.; Lastras-Montano, L.A.; "Improving read performance of Phase Change Memories via Write Cancellation and Write Pausing," In *Proc. of 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pp.1-11, 2010.

M. Saxena and M. M. Swift. FlashVM: virtual memory management on flash. In *Proc. of 2010 USENIX conference on annual technical conference (USENIX ATC '10)*, 2010.

X. Wu and A. L. N. Reddy. SCMFS: a file system for storage class memory. In *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pp. 1-11, 2011.

W. Zhang, T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures," In *Proc. of 18th International Conference on Parallel Architectures and Compilation Techniques*, pp. 101-112, 2009.

P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proc. of the 36th annual international symposium on Computer architecture (ISCA '09)*, pp. 14-23, 2009.