Plate: Persistent Memory Management for Nonvolatile Main Memory

Toshihiro Yamauchi,
Yuta Yamamoto, Kengo Nagai
Okayama University
3-1-1 Tsushima-naka, kita-ku,
Okayama, 700-8530 Japan
yamauchi@cs.okayama-u.ac.jp

Tsukasa Matono, Shinji Inamoto,
Masaya Ichikawa,
Masataka Goto
Kyushu University
6-10-1 Hakozaki Higashi-ku,
Fukuoka, 812-8581, Japan

Hideo Taniguchi
Okayama University
3-1-1 Tsushima-naka, kita-ku,
Okayama, 700-8530 Japan
tani@cs.okayama-u.ac.jp

ABSTRACT

Over the past few years, nonvolatile memory has actively been researched and developed. Therefore, studying operating system (OS) designs predicated on the main memory in the form of a nonvolatile memory and studying methods to manage persistent data in a virtual memory are crucial to encourage the widespread use of nonvolatile memory in the future. However, the main memory in most computers today is volatile, and replacing high-capacity main memory with nonvolatile memory is extremely cost-prohibitive.

This paper proposes an OS structure for nonvolatile main memory. The proposed OS structure consists of three functions to study and develop OSs for nonvolatile main memory computers. First, a structure, which is called *plate*, is proposed whereby persistent data are managed assuming that nonvolatile main memory is present in a computer. Second, we propose a persistent-data mechanism to make a volatile memory function as nonvolatile main memory, which serves as a basis for the development of OSs for computers with nonvolatile main memory. Third, we propose a continuous operation control using the persistent-data mechanism and plates. This paper describes the design and implementation of the OS structure based on the three functions on The ENduring operating system for Distributed EnviRonment and describes the evaluation results of the proposed functions.

CCS Concepts

• **Software and its engineering → Operating systems** • Software and its engineering → Virtual memory • Software and its engineering → Main memory

Keywords

Operating system, Persistent mechanism, Nonvolatile main memory, Memory management

SAC'16, April 4-8, 2016, Pisa, Italy.
Copyright 2016 ACM 978-1-4503-3739-7/16/04…$15.00.
DOI: http://dx.doi.org/10.1145/2851613.2851744

1. INTRODUCTION

A computer locates a program or data in the memory where the program is executed or the data are processed. However, the main memory in most current computers is volatile. Thus, the existing operating system (OS) or application programs (APs) store data in a volatile memory and make them persist in an external storage device (a nonvolatile storage medium). Ideally, accessing and processing of the persistent data and executing the program should be made within the memory.

Over the past several years, nonvolatile memory has actively been researched and developed [1]. Efforts have been made to mass produce nonvolatile memory in different forms such as PCM, MRAM [2], and ReRAM with quick access similar to DRAM. The technology to overcome the drawbacks of these memory forms has been studied in [3]–[5]. If these memory forms have adequate access speed, capacity, and price that allow them to replace DRAMs, then future computers can be equipped with a nonvolatile memory, and piece-by-piece writing of updated data to an external storage device, as currently performed, is not necessary. In addition, persistent data traditionally stored in an external storage device can be effectively managed using only the nonvolatile memory. Therefore, file input and output processing need not be done, and the OS and APs can use the persistent data by accessing only the memory, which could substantially improve the efficiency of the OS and AP processing.

Having a nonvolatile main memory in a computer would be preferable from the perspective of computer fault tolerance and process continuity. Therefore, studying OS designs predicated on the main memory in the form of nonvolatile memory that manages persistent data in a virtual memory are crucial to encourage widespread use of a nonvolatile main memory. However, replacing a high-capacity main memory with a nonvolatile memory is extremely cost-prohibitive. Thus, creating experimental conditions where a nonvolatile memory is used as the main memory is difficult.

This paper proposes a new OS structure for a nonvolatile main memory. In particular, this paper proposes three functions to study and develop OSs for nonvolatile main-memory computers. First, a structure, which is called *plate*, is proposed whereby persistent data are managed assuming that nonvolatile memory is installed in a computer. The plates allow the OS and APs to use persistent data by simply accessing a virtual memory. Data traditionally stored in files are made persistent by file operations. In the proposed OS structure, the persistent data are all mapped to a space in a virtual memory for the kernel. In addition, APs require that persistent data in the virtual memory for the kernel be mapped to a virtual memory for each AP to use these data. Thus, processing of the files is not necessary. We assume that a 64-bit address space is used for the virtual memory; thus, all persistent data can be mapped to virtual address spaces.

Second, we propose a persistent-data mechanism to serve the volatile main memory as a nonvolatile main memory. This mechanism serves as a basis for the development of OSs for computers with a nonvolatile memory. In this mechanism, all memory areas in the virtual memory for the kernel and user are made persistent. This mechanism is required to implement and evaluate the plate.

Third, we propose a continuous operation control using the persistent-data mechanism and plates, which can save and restore the state of OS and AP processing before and after computer reboots. This control assumes that OS- and AP-related data are retained in the nonvolatile memory. Even if the processing stops, e.g., when the computer power supply is disconnected during OS or AP processing, the continuous operation control uses the data in a nonvolatile format to resume the OS and AP processing. This function is implemented based on the persistent-data mechanism and plates mentioned earlier.

This paper describes the design and implementation of the plates on The ENduring operating system for Distributed EnviRonment (***Tender*** OS) [6], which we have developed for 21 years. This paper also describes the results of the evaluations of the proposed functions.

2. PLATE: PERSISTENT-DATA MANAGEMENT FOR NONVOLATILE MEMORY

2.1 Design of Plate

A structure is proposed whereby persistent data are managed assuming that nonvolatile main memory is installed in a computer. In existing OSs, persistent data are managed using external storage. On the other hand, the proposed structure allows the OS and APs to use persistent data by simply accessing a virtual memory. We call this persistent-data management *plate*.

The relationship between the plates and a virtual address space is shown in Figure 1. The memory regions in the virtual address space for the kernel and APs that can be accessed are created by a request of the OS or APs. A plate function manages all the memory regions in the virtual address space as plates. Thus, the plate interface must be compatible with the existing memory allocation and deallocation interfaces. For example, where a memory region is required for data that are temporarily used for the execution of a process, a plate is created and used in the virtual address space of the process. The plate is then deleted after it has been used. Figure 1 shows the text segment, data segment, stack segment, and data2 as examples of the data in the virtual address space of the upper left process.

The metadata of the plate are managed in the kernel virtual address space. The metadata of the plates include the top address, size, position of the persistent area in an external storage, plate name, resource identifier, owner id, and access right where the main memory is volatile. If a nonvolatile memory is used as the main memory in a computer, the top address of the persistent data is included, instead of the persistent area position in an external storage. The metadata of the plate are similar to the inode of a file system, but the inode does not include a top address. The plate provides namespace for the kernel and APs as plate names. This structure can provide hierarchical namespace of the plates such as file name and directories of the file systems. In addition, the metadata of the plates always exist in the kernel virtual address space to access and manage the metadata and contents of the plates by the kernel. In contrast, inode is basically stored in an external storage.

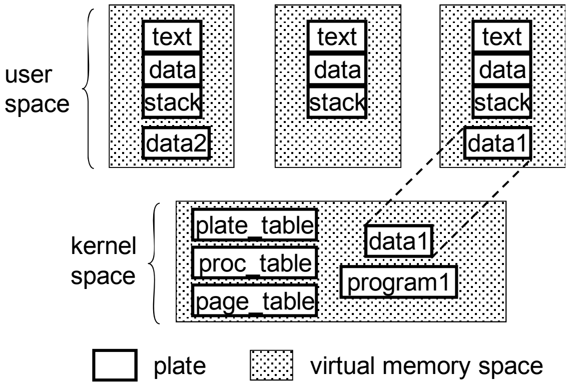

Figure 1 Relationship between plates and virtual memory space.

When an AP uses the persistent data in a plate, it issues a system call that attaches the requested plate in the kernel virtual address space to the AP virtual address space, as shown in data1 in Figure 1. The function of this system call is similar to the open system call of existing OSs. Moreover, when the process completes the use of this plate, it issues a system call that detaches the plate from its virtual address space. This system-call function is similar to the close system call.

2.2 Interface for persistent data

In the plate structure, six interfaces (creation, deletion, attachment, detachment, right-to-access change, and resizing) are present, and all interfaces are comparable with that of the memory operation interfaces provided in existing OSs. The purpose of the six interfaces is described as follows: the functions equivalent to the creation and deletion of a memory region are provided to make all memory regions persist using a plate structure. The attachment and detachment functions are used to utilize the plate in the virtual address space of the APs. The right-to-access change function protects the plate by changing its access right.

In addition to these interfaces where the main memory is volatile, a write function is provided to write the contents on a plate to an external storage for the plate to persist. The target of the write function is a specific plate or all the plates.

2.3 Restoration process after rebooting where the main memory is nonvolatile

The restoration process of a nonvolatile main memory and plates are performed during the OS startup process. After the boot loader loads the OS kernel, the OS start process is run. First, the OS start process sets up the essential hardware and enables the virtual memory. Next, it initializes the device drivers. Then, it restores all the plates using the metadata information in the management data for the plate in the nonvolatile main memory. Completion of the restoration of all plates indicates that all OS and AP contents in the virtual address space are also restored, as shown in Figure 1. Finally, the continuous operation control (described in Section 4) dispatches a process running during the preservation process. We assume that the information on the registers is stored in the nonvolatile main memory during the storing process, and the register information can be used in the dispatch process. Then, all the OS and AP processes can be resumed.

3. PERSISTENT-DATA MECHANISM

3.1 Challenges

We propose a persistent-data mechanism to make a volatile main memory function as a nonvolatile main memory. This mechanism provides the property of persistence in all volatile main memory regions. Thus, the OS and APs can operate the main memory as a nonvolatile memory by simply accessing the virtual memory.

3.2 Design

The persistent-data mechanism is designed as a function of the OS. The persistent-data mechanism manages the volatile main memory to save the contents of the volatile main memory to external storage areas.

The persistence attribute is provided to the volatile main memory by a persistent function (described later), and it is attached to an external storage area. Therefore, the data exist in a volatile main memory or an external storage, and each plate in the volatile main memory is attached to an area in an external storage. Then, the content of the volatile main memory can be written to the area in the external storage for it to persist from a write-operation request.

An area of the volatile main memory is mapped in the volatile main memory and in an external storage when the memory area is created, as shown in Figure 2. The persistent function creates the stored region in the external storage when the memory region is created in the virtual address space. This storage region in the external storage is called the persistent area. A file-system partition can be used as a persistent area.

Even if neither the OS nor the APs perform write operations, the contents of the volatile main memory are written to an external storage from the request of the continuous operation control (described later). In other words, the persistent operations are transparent against the OS and APs. In addition, the persistent function can write when a process is created or deleted and upon the request from the OS or APs. During such process, only the updated pages are written in the persistent area in an external storage. The OS and APs can also designate a memory region (plate) or all memory regions (plates) to be the target of the write operation. As described earlier, other functional blocks of the OS do not need to be aware of the data writing from the volatile memory to an external storage.

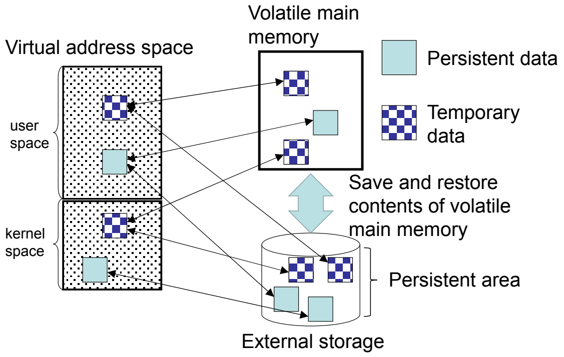

Figure 2 Data mapping of volatile main memory and persistent data on proposed scheme.

Furthermore, the same interface can be provided as a memory region operation for memory (plate) operations because the persistent-data mechanism provides the persistent function without modifying the OS and APs. Additionally, this structure can be implemented if a computer supports a virtual memory.

3.3 Target of persistent memory

In many OSs, the temporary data stored in the main memory can be transferred to an external storage if a page out is required. Swap partition can be used for external storage, but in this work, we use a file system. The persistent data in a plate are stored in an external storage as a file.

The proposed structure aims to resume the processing of the OS and the APs. For this reason, the targets of persistent operations are all the plates, including all temporary and persistent data that exist in all the virtual address spaces. These regions include the region of a text segment, data segment, stack segment, and management data for the OS, in addition to all segments of the APs.

4. CONTINUOUS OPERATION CONTROL FOR OS AND AP PROCESSING

In case the main memory is volatile, the continuous operation control can restore the same contents of the volatile main memory regions at the time of preservation after rebooting the computer based on the data saved in the external storage before the OS restarts. As a result, the OS and the APs can use the volatile memory as a nonvolatile memory.

First, we explain the flow of the persistent process. The continuous operation control can store all the contents of the volatile main memory to an external storage using the persistent function. Persistent processing of the volatile main memory can be performed during the OS termination processing or cyclic write processing or by a request for write operation from the OS and APs. Specifically, the data in all the volatile main memory mapped to the plates in the virtual address space are written to the persistent area in an external storage. Next, the metadata that manage the plates must be written to the designated position in an external storage in the persistent processing to maintain coherence between the metadata and the plate.

The restoration process of the nonvolatile main memory and plates are performed during the OS startup process. After the boot loader loads the kernel of the OS, the OS start process is run. First, the OS start process sets up the essential hardware and enables the virtual memory. Next, it initializes the device drivers and sets up the file-system controllers. Then, in the OS start process, the metadata of the plates are loaded into a memory region from the designated position in an external storage. Instead of directly restoring the saved contents of the volatile main memory from an external storage, plates related to the memory management, such as page table, are first restored. After all page directories and page tables are restored, the virtual address space is also restored. Then, the other plates are restored in the virtual address spaces using the information on the metadata in the management data for the plate. Before restoration of the other plates, memory regions are created on the same address and with the same size in the virtual address space. Next, the contents of the plates are loaded into the memory regions from the persistent data in the external storage. This process means that the same contents in the volatile main memory regions at the time of preservation are restored. Then, the OS startup process executes the initial processing of resources whose target includes the uninitialized resource manager and the OS components. Finally, the continuous operation control dispatches a process that is running during the preservation process. We assume that the information on the registers is stored in an external storage during the storing process, and the register information can be used in the dispatch processing. Then, all the OS and the AP processing can be resumed.

5. IMPLEMENTATION

5.1 Integration of plate structure

We implemented the plate structure in the ***Tender*** OS. The plate structure provides memory management interface, instead of the existing memory management interface of the OS. It provides persistent memory regions to the OS and APs using the memory management functions in the OS. In addition, because we assumed a volatile main memory in this research, we implemented the persistent-data mechanism and continuous operation control in the ***Tender*** OS.

We now present a simple description of the ***Tender*** OS, an operating system that implements the proposed structure. We also provide a simple overview of the memory and process resources of the ***Tender*** OS related to this work.

5.2 *Tender* OS

In the ***Tender*** OS [6], we encapsulate the objects manipulated by the OS as resources and separate them so that they become independent. We assign a resource name and resource identifier to each resource and unify the interface to manipulate the resources. The resource identifiers and resource names include location information that indicates a particular machine. The resource names are managed by a tree structure. An example of a resource name is "/machine1/process/procA".

The interface for the operation of resources is unified. Program components that operate both the local and remote resources are called through a unified interface. The unified interface is named as resource interface controller (RIC). The RIC has a pointer table that contains all pointers of the program components. The program components consist of five programs, namely, open, close, read, write, and control. Each program component must call the RIC to call any program components. Bypassing the RIC is prohibited in the ***Tender*** OS kernel.

Additionally, we also separate the management information for individual resources on a per-resource level and forbid references among management tables for each resource. The existence of an individual resource does not depend on the other resources, including the processes, because the management table for each resource is separate, i.e., each resource can exist irrespective of the existence of other resources. In this manner, by making the resources separate from one another and independent, a fast process creation and termination mechanism [7] using recycling process and memory resources is proposed in the ***Tender*** OS.

5.3 Memory and process management

A process is composed of various process components. The process components in the memory space of the ***Tender*** OS are called process resources. A process is divided into six types of resources, namely, process, virtual region, virtual space, virtual user space, virtual kernel space, and physical memory, in the ***Tender*** OS.

The virtual region is a resource that virtualizes the data storage region, which is mapped to the physical memory or external storage. It contains information about the storage area, which is in the physical memory or in an external storage, in its management table. The virtual space is a space for the virtual address and corresponds to the page table where a virtual address is mapped to a physical address. The virtual user space is a space accessible from the processor by the virtual address. It is created by attaching the virtual region to the virtual space and is deleted by detaching. Here, attaching means storing the information in the data storage region as an entry in the page table.

5.4 Plate resource

Given that a plate is the unit of an object controlled and managed by the OS, the plate is implemented as a resource in the ***Tender*** OS. One plate consists of a virtual kernel space, a virtual user space, a virtual region, a physical memory, and a persistent area in an external storage.

The persistent-data plate is created in the kernel virtual memory by the plate-creation function. The OS can use the plate that exists in the kernel virtual memory, and all processes can access this persistent-data plate by attaching to the plate on the user virtual memory.

In addition, in the ***Tender*** OS and in order to enable data transfer among different OSs, the data stored in the external storage of a plate are a file format of the existing OS. However, the plate management does not directly operate a file, but it can operate the persistent unit resource.

5.5 Interface of the plate resource

The seven interfaces provided by the plate management to the APs and to each functional block of the OS are listed in Table 1. In the case of the plate resources, the open operation is create_plate, the close operation is delete_plate, the read operation is attach_plate, the write operation is detach_plate, and the remaining three operations are control operations. A plate name is included in the resource name and can be used similar to a filename.

A plate is created in a virtual address space for kernel by the create_plate function. The OS can use the plate. Before the APs use the plate, they call the attach_plate function to attach the plate to their virtual address space. Then, the APs can access the plate. After the use of the plate is finished, the plate is detached by the detach_plate function. In case a plate is unnecessary, it is deleted by the delete_plate function.

In the change_prot_plate function, access permission to the data in a plate is changed to read only or to read and write. The size of the plate is changed by the change_size_plate function. Moreover, in the persist_plate function, the updated memory regions are written to the persistent area of an external storage.

Table 1 Plate management interface.

Interface	Function
create_plate(name, access, etc.)	Creates the plate specified by parameters *name*, *access* right, and so on, and returns *plateid*, which is a resource identifier.
delete_plate(plateid)	Deletes the plate *plateid*. It also releases all memory areas and persistent areas from the plate's external storage.
attach_plate(plateid, vmid, addr, access)	Attaches the plate *plateid* to the virtual address space *vmid* specified by the attached address *addr* and the *access* right.
detach_plate(plateid, vmid, addr)	Detaches the plate *plateid* from the virtual address space *vmid*. The address of the plate is specified by *addr*.
change_prot_ plate(plateid, access)	Changes the access right of *plateid* to *access*.
change_size_ plate(plateid, addr, size)	If size > 0, an area with a size of *size* is inserted into the *addr* of the *plateid*. If size < 0, an area with a size of *size* is released from the *addr* of *plateid*.
persist_plate(plateid)	Writes the updated data of *plateid* into an external storage.

The write operation can be called from the OS or APs. A daemon program can also periodically call the write operation using the central processing unit (CPU) idle time. This method can be deployed using a periodical timer resource in the ***Tender*** OS. Moreover, by synchronizing the writing process with the deletion of a process, the results of such processing can also certainly be persisted by writing.

5.6 Persistent unit resource

Persistent unit is a resource abstracted from the persistent area of the external storage, and it conceals the data stored form of a persistent area to a plate manager. A persistent unit is mapped to a persistent area of the external storage. One of the existing file systems can be used as data stored in an external storage. For example, the fast file system (FFS) of the Berkeley software distribution, the new technology file system (NTFS) of Windows, and the Ext2 file system (Ext2fs) of Linux can be used.

During the creation, the plate management creates a persistent unit and attaches the plate to the persistent unit. The persistent unit is also mapped to a persistent area in an external storage. Thereby, the plate is continued using a persistent unit. In addition, in the present ***Tender*** OS, FFS [8] is implemented as the file system of an external storage.

5.7 Limitation

The persistent-data mechanism makes the plates persist using a write request from the OS and APs. If a computer power supply is disconnected during OS or AP processing, the updated data on the volatile memory are lost. On the other hand, if a nonvolatile memory is deployed, all data become persistent, and no data are lost. However, the persistent-data mechanism can resume processing of the OS and APs based on the stored data on the external storage. Thus, the damage due to data loss can be reduced.

6. EVALUATION

6.1 Purpose of evaluation

First, we evaluate whether the proposed three functions can save and restore the contents of the plates and whether it can continue the processing of the OS and APs or not. We also measure the processing time of the write operations. We then evaluate the influence on the other processes, and the effect of updating only the pages are written during the write operations. Then, we evaluate the processing time of the plate restoration. Finally, we evaluate the influence of the cyclic write operations on the other processes.

To demonstrate the feasibility of the proposed method, we performed the following four evaluations of the proposed mechanism on the ***Tender*** OS.

(Evaluation 1) Continued operations of the OS and AP processes

(Evaluation 2) Writing the plate operation

(Evaluation 3) Plate restoration process

(Evaluation 4) Cyclic writing method by considering the CPU idle time

The evaluations were performed using a computer (CPU: Celeron D 2.8 GHz, HDD: 7200 rpm Ultra ATA/100, OS: ***Tender*** OS). The number and total size of the plates that existed when Evaluations 1, 2, and 3 were performed are listed in Table 2.

Table 2 Number of plates and total size during the evaluation.

Number	Number of plates			Plate total size (KB)		
	Kernel	User	Total	Kernel	User	Total
1	159	22	181	6,504	632	7,136
2	143	12	155	6,428	224	6,652
3	143	12	155	6,428	224	6,652

6.2 Continued operations of the OS and AP processes

We evaluated whether the ***Tender*** OS and APs processes can be continued when running an imprecise computational program [9] that calculates an approximate solution for natural logarithms. The program consists of two processes. One process calculates an approximate solution. The other process receives the approximate solution from the calculation process.

Simultaneous with the running of the AP mentioned in the previous section, another program issues a write operation that writes all the plates. Then, the contents of all the plates that persisted are written to an external storage. To evaluate whether the OS and AP processing would continue after rebooting the test computer, the power supply was suddenly disconnected, and the computer was rebooted after reconnecting the power supply.

The persistent function can write all the contents in the volatile main memory to a persistent area when the write operation is called as a result of the evaluation indicated in the previous paragraph. Moreover, during the restoration process at the OS startup, each plate was restored in all the virtual address spaces, and all resources that the OS manages could be restored. All the processes could continue working after the reboot. Furthermore, the imprecise computational program previously described also normally resumed its processing from the reboot. The OS and its resources continued their normal processing after the computer rebooted.

Moreover, in the other experiments, all the resources managed by the OS were restored during the OS startup. We verify that the state of all the resources was completely restored and that all the processes in the ***Tender*** OS continued after the reboot in the experiments. For example, the execution resource possesses a degree that can be assigned to a processor [10]. The amount of time assigned to an execution is determined by the execution degree. In the experiments, all the processes were normally restored as well as the scheduling queue in the execution management table. Then, all the scheduled processes were resumed.

6.3 Plate write operation

To evaluate the write performance of a plate, the data size of the write process, the number of plates, and the processing time were measured when the system call that wrote on all the plates immediately after the OS startup process was finished. Moreover, the system call that performed the write operation of all the plates was issued once again immediately after the previous system call was issued. The evaluation results are listed in Table 3.

The first write data size evaluated was 641 KB, and the total size of its existing plate was 6,652 KB, which means that the plate-write process searched the page updated in the memory and then wrote to an external storage. Given that only approximately 10% of the plates in the memory were updated among all the plates in this evaluation, only few of the plates were written in this experiment. Moreover, considering that few of the memory regions were updated by the second write process, the number of pages written in the second write process was small compared with the first write process, which shows the validity of the function that only writes the updated pages.

Table 3 Processing time of the second evaluation.

	Write-data size (KB)	**Number of plates**	**Processing time**
First	641	53	359.4 ms
Second	159	22	96.3 ms

6.4 Plate restoration processing

After the write process from Evaluation 2, the processing time was measured when the computer was rebooted and the plate was restored in the ***Tender*** OS startup process. The processing time of the plate restoration was measured.

During the OS startup process, the plate-restoration processing time was measured after the end of the initialization process of the device. The processing time was 3,883.8 ms. During the plate-restoration process, after all the plates have been restored in the virtual address space of a kernel, a plate was also restored in the virtual address space for each user. In this case, all plates (a total of 6,652 KB) were loaded from the external storage, and all were restored in the virtual address spaces. In addition, during the restoration process of the current implementation, demand paging was not adopted, but the OS allocated a physical memory for all the plate areas in the virtual address spaces. The plate-restoration process repeatedly issued read requests to the persistent unit manager. Therefore, the read processing time from an external storage occupied a greater portion of the plate-restoration process. The read processing time in the restoration can be reduced by deploying demand paging.

6.5 Cyclic write method by considering the CPU idle time

The overhead during the write process was periodically evaluated. In this measurement, two programs were run. The first program was AP 1, which periodically issued the write system call; AP 1 was run by a kernel mode. The second program was AP 2, which repeatedly wrote a value to a plate. The write cycle of AP 1 was changed from 1 to 16 min, and the write-data size was also changed. The write processing time and the total size of the written data were measured for 32 min. The size of the memory area created by AP 2 was 2 MB. The following results are listed in Table 4.

1. The size of the written data was approximately 2 MB per time, and the updated data of AP 2 occupied most of the written data.
2. The total write processing time increased according to the write cycle.
3. The write processing time where the cycle was 16 min was approximately 0.09% of the write cycles. Thus, the time was small.

The overhead of the proposed method was less than 10% when the write cycle was longer than several minutes.

Table 4 Evaluation results of the cyclic write function.

Cycle of write (min)	1	2	4	8	16
Number of write	32	16	8	4	2
Total size of write data (KB)	69,862	34,931	17,466	8,733	4,366
Total processing time (s)	26.23	13.51	7.66	3.40	1.70
Processing time per second (s)	0.82	0.84	0.96	0.85	0.85

7. RELATED WORKS

Many computers have main memory in the form of volatile memory. Thus, if the power supply to a computer with an existing OS is stopped, then the computer shuts down abnormally.

Individual APs may fail to make data persistent. In addition, existing OSs only make some of the data used by APs persistent. Thus, APs processing that is underway cannot be restored even if the computer is rebooted.

The hibernation mode in Windows and the hibernation mode via swsusp or TuxOnIce [11] in Linux allow computer processing to be interrupted and resumed. The hibernation mode writes all of the data in main memory to an external storage device and it loads saved data into the main memory once the computer is rebooted. This allows processing to be resumed after computer processing has been interrupted. However, a computer has to be put into hibernation by a user. This is effective when a halt is foreseeable but fails to address an unforeseeable emergency shutdown.

Some OSs also have a checkpoint restart capability [12], [13]. This capability saves and restores the state of processing of individual APs. Thus, this capability can only be used to restore processes that are part of an AP. This capability is unable to save or restore the state of an OS of the state of APs. Membrane [14] allows restoration from a checkpoint in the event of a file system error. Thus, APs need not account for the occurrence of file system errors, and operations can continue. However, this capability cannot deal with a computer shutting down as a result of termination of the power supply.

KeyKOS [15], [16] and EROS [17] have a checkpoint and a restart mechanism. All the data and processes are checkpointed. These OSs are microkernels, and the data used for the checkpoint is stored in the disk regions called the checkpoint area. These OSs and ***Tender*** OS differ from each other. ***Tender*** OS is a monolithic kernel. In addition, in order for a persistent-data mechanism provides the property of persistence in all volatile main memory regions, all the data on the virtual address space are stored in an external storage.

Grasshopper [18], [19], [20] supports persistent processes. Containers are the only storage abstraction provided by Grasshopper; they are persistent entities that replace both the address spaces and the file systems. However, although Grasshopper provides persistence of processes, it cannot provide persistent processing of the OS.

Guerra et al. [21] proposed software persistent memory (SoftPM). The SoftPM persists a persistent memory abstraction called container. Volos et al. [22] also proposed Mnemosyne: lightweight persistent memory. Our proposed structure can persist all memory area in virtual memory and serve the volatile main memory as a non-volatile main memory.

Li et al. [23] proposed NV-process, a fault-tolerance process model based on NVRAM. NV-process instances run in a self-contained way in NVRAM, thus to survive across OS reboot. However, only NV-process can continue running when OS reboots. By contrast, the proposed structure can save and restore the state of OS and AP processing.

Condit et al [24] presented a file system and a hardware architecture that are designed around the properties of persistent, byteaddressable memory. One of the design principle of this work is to allow OSs and APs to easily exploit the benefits of fast, byte-addressable, nonvolatile memory for file system. Lee et al. [25] proposed a buffer cache architecture that subsumes the function of caching and journaling in a unified nonvolatile memory space. Moraru et al. [26] proposed a new memory allocator for NVRAM can ease the task of creating safe, high-performance persistent data structures for emerging nonvolatile memories.

Bailey et al. [27] discussed the OS implications of new NVRAM in future systems. They discussed four system architecture options for NVRAM such as hybrid main memory and all-NVRAM. Mogul et al. [27] have investigated OS support for hybrid main memory. Our target is all-NVRAM option. This option will affect significantly design of OS in future. Therefore, we pursue new OS designs for all-NVRAM main memory.

8. CONCLUSIONS

This paper has proposed three functions to study and develop OSs for nonvolatile main memory computers. First, we proposed a new OS structure *plate* whereby persistent data are managed assuming that nonvolatile memory is present. The proposed structure allows the OS and APs to use the persistent data by simply accessing the memory. Second, we proposed a persistent-data mechanism to make the volatile main memory function as a nonvolatile main memory. We showed that the development of an OS and APs in this environment allows the development of an OS and APs predicated on the main memory in the form of a nonvolatile memory.

Third, we proposed the continuous operation control using the persistent-data mechanism and a plate. This control can save and restore the state of OS and AP processing. The continuous operation control assumes that the OS- and all AP-related data are retained in the nonvolatile memory. Even if the processing stops, such as when the computer power supply is disconnected during the OS or APs processing, the continuous operation control uses the data in a nonvolatile format to resume the OS and AP processing.

Moreover, the evaluation results showed that each memory region was restored on all the virtual address spaces during the restoration process at the OS startup, and all the resources managed by the OS could be restored. All processes could continue working after the reboot. The evaluation results showed that the write process time of the plate is proportional to the size of the updated pages that will be written, and the read processing time from an external storage occupies a larger portion of the plate restoration process.

These results show that the proposed mechanism can be a basis for the development of OSs for computers with a nonvolatile memory. In addition, the results show that the plate and the continuous operation control can be used for computers with a nonvolatile main memory.

9. REFERENCES

[1] B. C. Lee, et al. Phase-change technology and the future of main memory. *IEEE Micro*, Vol.30, Issue 1, pp.131-141, 2010.

[2] J. Heidecker. MRAM Technology Status. *JPL Publication*. 13-3, 2013.

[3] B. C. Lee, 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.

[4] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory

technology. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*, pp.14-23, 2009.

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

[6] ***Tender*** project, http://www.swlab.cs.okayama-u.ac.jp/lab/tani/research/tender-e.html

[7] T. Tabata, H. Taniguchi. An improved recyclable resource management method for fast process creation and reduced memory consumption. *International Journal of Hybrid Information Technology*, Vol. 1, No. 1, pp.31-44, 2008.

[8] M. K. McKusick, W. N. Joy, S. J. Leffler, R. S. Fabry. A Fast File System for UNIX. *ACM Trans. Computer Systems*, vol.2, no.3, pp.181-197, 1984.

[9] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, vol.31, pp.314-333, 1998.

[10] T. Yamauchi, T. Hara, H. Taniguchi. A mechanism for achieving a bound on execution performance of process group to limit CPU abuse. *The Journal of Supercomputing*, Vol.65, Issue 1, pp.38-60, 2013.

[11] N. Cunningham. TuxOnIce. http://tuxonice.nigelcunningham.com.au/ .

[12] O. Laadan, J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of 2007 USENIX Annual Technical Conf.*, pp.323-336, 2007.

[13] S. Yi, J. Heo, Y. Cho, J. Hong. Adaptive Page-level Incremental Checkpointing based on Expected Recovery Time. In *Proceedings the 2006 ACM Symposium on Applied Computing*, pp.1472-1476, 2006.

[14] S. Sundararaman and S. Subramanian and A. Rajimwale and A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseauand and M. M. Swift. Membrane: Operating System Support for Restartable File Systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. 281-294, 2010.

[15] N. Hardy. The KeyKOS Architecture. *Operating Systems Review*, vol.19, No.4, pp.8-25, 1985.

[16] A.C. Bomberger, A.P. Frantz, W.S. Frantz, A.C. Hardy, N. Hardy, C.R. Landau, J.S. Shapiro. The KeyKOS Nanokernel Architecture. In *Proceedings the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp.95-112, 1992.

[17] J. S. Shapiro, J. M. Smith, D. J. Farber. EROS: a fast capability system. In *Proceedings 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pp.170-185, 1999.

[18] A. Lindstrom, R. di Bona, A. Dearle, J. Rosenberg, F. Vaughan. Persistence in the Grasshopper Kernel. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, ACSC-18, pp 329-338, 1995.

[19] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindstrom, J. Rosenberg, and F. Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Computing Systems*, Vol.7, No.3, pp. 289-312, 1994.

[20] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris. Operating system support for persistent and recoverable computations. *Communications of the ACM*, Vol. 39, Issues 9, 62-69, 1996.

[21] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12)*, 2012.

[22] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS XVI)*, 2011.

[23] X. Li, K. Lu, X. Wang, X. Zhou. NV-process: A Fault-Tolerance Process Model Based on Non-Volatile Memory. In *Proceedings of the Third ACM SIGOPS Asia-Pacific conference on Systems (APSys'12)*, 2012.

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

[25] E. Lee and H. Bahn, S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, 2013.

[26] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, N. Binkert, P. Ranganathan. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of ACM Conference on Timely Results in Operating Systems (TRIOS)*, 2013.

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

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