Shadow Block: Imposing Block Device Abstraction on Storage Class Memory*

Jaemin Jung Jeongheon Choi Youjip Won Sooyong Kang
Division of Electrical and Computer Engineering, Hanyang University
Seoul, Korea ZIP 133-791
{jmjung, rhythmical, yjwon}@ece.hanyang.ac.kr sykang@hanyang.ac.kr

ABSTRACT

Storage Class Memory is a dream device. Once realized into proper scale, it will resolve the hassles which current storage system suffers from. In this work, we focus our effort in devising a method to impose block device abstraction on Storage Class Memory. Block I/O on Storage Class Memory needs to be performed in atomic fashion and "partial write" on Block I/O should be prohibited. We propose "Shadow Block" to provide atomicity of block operation in Storage Class Memory. We perform out-of-place update on Storage Class Memory block and use block mapping table to locate the actual block. To minimize the overhead of accessing block mapping table, we develop new path for `read` and `write` operation. We implement the new I/O subsystem, called Shadow Block, on embedded system with 64 Mbit FRAM. According to our experiment, the overhead of supporting atomicity is 0.1% and 1% for read and write operation, respectively.

Categories and Subject Descriptors

D.4.2 [**Operation Systems**]: Storage Management

Keywords

Atomic I/O, Block Device, Storage Class Memory

1. INTRODUCTION

Storage Class Memory is next generation memory device which can preserve data without the supply of electricity and which can be accessed in byte-granularity. Its access latency is expected to be comparable to that of DRAM and to be much faster than existing Flash based storage device, e.g. NAND Flash and NOR Flash. There exist several semiconductor technologies for Storage Class Memory. They include PRAM(Phase Change RAM), FRAM(Ferroelectric RAM), MRAM(Magnetic RAM), RRAM(Resistive RAM), Solid Electrolyte[4], SPIN-RAM[17] and etc. Each of these technologies has its own pros and cons. PRAM has the highest bit density, but still suffers from heat dissipation issue involved in changing the state of the cell. FRAM is very fast but it suffers from low bit density. It seeks its opportunity in small scale embedded device. It is currently too early to determine which of these semiconductor device will eventually survive in the market. However, the advancement of Storage Class Memory is going to resolve significant fraction of the technical hassles which current storage system suffers from: reliability, heat, power consumption, and not to mention speed.

Storage Class Memory can be used as a memory as well as a storage. This characteristic holds profound implication on Operating System. I/O intensive workload can mutate to CPU intensive workload if data resides in Storage Class Memory instead of legacy storage device, e.g. HDD. Current Operating System draws clear line between memory and storage and handles them very differently. Memory system and storage system is accessed via address space and via file system name space, respectively. In terms of latency, scale, I/O unit size and etc., memory and storage dwell in entirely different world from Operating Systemą́s point of view. Operating System uses `load-store` and `read()/write()` interface for memory and storage device, respectively. The method to locate object and the method to protect the object against illegal access are totally different in memory and storage device. Advancement of Storage Class Memory now calls for redesign of various operating system techniques, e.g. context switch, paging mechanism, read/write, protection and etc., to effectively exploit its physical characteristics. Table 1 summarizes the characteristics of memory, storage, and storage class memory.

Given the access speed of the Storage Class Memory, it can be integrated with memory address space via standard SRAM interface or high speed I/O interface, e.g. PCI. It is also possible that it is packaged using legacy HDD like foam factor as in the case of SSD(solid state disk) and connected to host via standard I/O interface, e.g. SCSI, SATA or FC-AL. We are interested in the former case where storage class memory and DRAM forms homogeneous memory address space and where storage class memory is accessed via memory address as well as via block device.

In this work, we focus our effort on one of the most fundamental issues in I/O subsystem: *How to impose Block Device Abstraction over Storage Class Memory?* The key ingredient of this issue is an ability to write block on Storage Class Memory in *atomic* fashion. For block device, it is mandatory that write operation is performed in block unit and that partially written block does not exist. Writing a block into memory can partially complete when the system

*This research is supported by KOSEF through National Research Lab(ROA-2007-000-20113-0) at Hanyang University, Korea and Samsung Electronics.

	Domain	Access Path	Unit	Protection	Overhead
Memory	Address Space	load/store	word(e.g. 4 Byte)	page table	N/A
Storage	Name Space	read/write	block(e.g. 4 KByte)	offset check, ACL	system call
SCM	Both	Both	Both	unknown	unknown

Table 1: Memory and Storage(SCM: Storage Class Memory)

stops due to unexpected failure, e.g. power failure, buggy device driver or etc. Imposing block device abstraction on memory address space is not new. There exist a number of memory based file systems which use memory address space as file system partition, e.g. ramfs[15] and tmpfs[13]. Since contents of main memory are automatically reset when system reboots, these memory based file systems do not worry about the partial write issue. Storage class memory retains partially written block unless it is explicitly reset. Further, if the Storage Class Memory is used as storage, we cannot arbitrarily reset the respective region. However, it is not feasible to use existing file system techniques, e.g. journaling or log-structured file system to avoid partial write situation in storage class memory. Writing journal and data block will double the memory traffic and subsequently doubles the I/O latency. Appending ECC, parity, or signature to each block[6] can detect the partial write situation but cannot fix it. Imposing block device abstraction over Storage Class Memory is not trivial issue which requires elaborate treatment. In this work, we develop Shadow Block to support block level I/O in Storage Class Memory.

2. BLOCK I/O AND STORAGE CLASS MEMORY

2.1 Block Device and Atomicity of Write

Each layer in system hierarchy, e.g. memory, and storage, has its own notion of unit of operation. Minimum unit of memory access is defined by its Instruction Set Architecture and usually *word*. From DRAM's point of view, minimum unit of access is cache line. Storage device has its own I/O unit. Hard disk drive has a notion of *sector*(512 Byte or 2048 Byte) and Flash storage has a notion of *page*(512 Byte or 2048 Byte). File system has its notion of file system *block*. The key issue behind the notion of I/O unit is that that the respective layer should guarantee the *atomicity* on I/O operation. In hard disk drive and solid state drive, controller is responsible that sector(or page) is written in atomic manner. They append ECC to all sectors(or pages) to verify that a given sector(or page) is written correctly. Some of the high-end HDD's and SSD's adopt super capacitor so that controller cirtuitry can sustain its electric current in case of unexpected power failure until it finishes ongoing write operation.

Storage Class Memory can be integrated into the system via existing DRAM interface or high speed interface with byte addressability, e.g. PCI. Under this system architecture, the minimum unit of I/O on Storage Class Memory will be *word* which is much smaller(4 Byte) than the I/O block size(4096 Byte). Without the assistance of special hardware architecture, `write()` operation will be done via multiple `load-store` operations or via DMA. Existing Operating System and CPU architecture does not provide atomicity across multiple `load-store` operations. It is not sufficient to protect multiple load-store operations using existing synchronization primitive, e.g. `test-and-set`, semaphore or disabling interrupt. System needs to retain the old value until the block region of Storage Class Memory is completely updated.

2.2 Partial Write in Storage Class Memory

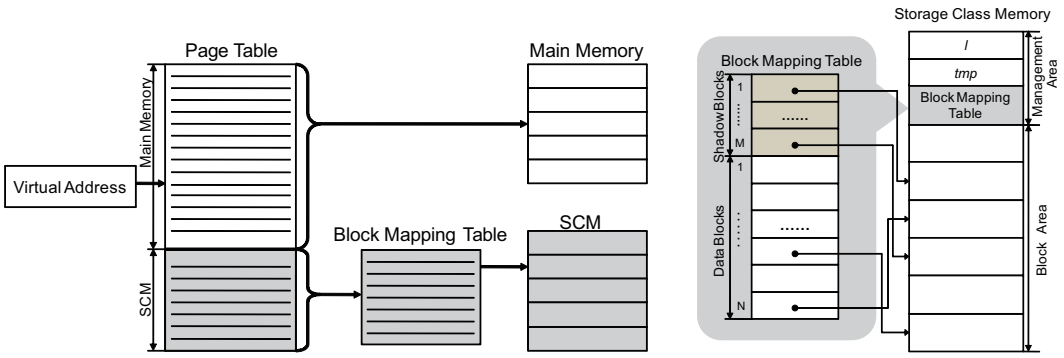
When Storage Class Memory is used as *storage*, Operating System accesses Storage Class Memory using `read()` and `write()` interface. This interface is fundamentally grounded at block device abstraction. Block I/O operation on Storage Class Memory raises important technical concern which requires in-depth elaboration. For file system to behave correctly, it is mandatory that individual I/O operation is performed in atomic fashion. Since storage class memory is a memory, block I/O on Storage Class Memory region will be done via the repetition of `load-store` instruction or DMA. However, different from the approach taken by existing main memory file system, memory copy of a block(e.g. 4096 Byte) needs to be performed in atomic fashion for Storage Class Memory. Otherwise, partially written block persists which can lead to the file system corruption.

3. SHADOW BLOCK

3.1 Design

In this work, we develop a mechanism called 'Shadow Block' to provide atomicity on block I/O over Storage Class Memory. Shadow Block shares the same idea with shadow paging which was designed to provide atomicity in database transaction[2]. In shadow paging, database transaction does not update the original copy. Instead, it creates and updates shadow page and shadow page table. Once transaction completes, shadow page table replaces the existing page table. In shadow paging, several versions of unoriginal page coexist if multiple transactions accesses the same page in concurrent manner. This is because single transaction can be active for significant amount of time and it is not desirable to give an exclusive lock on a given page for an entire time span during which that transaction is active. Multiversion concurrency control is important issue in shadow paging[16]. In Shadow Block, however, copying a block of memory is quick operation and that it is not necessary to address multiversion issue in our Shadow Block environment. This argument will be verified through physical experiment in section 4.3.

There exist two ways to integrate physical address of Storage Class Memory to Virtual Memory: *fixed binding* and *dynamic binding*. In fixed binding, a ceratin address space in VM is permanently bound to Storage Class Memory. In dynamic binding, VM address can be mapped to either DRAM or Storage Class Memory dynamically. While dynamic binding provides more flexible address space management, we carefully argue that it is far more complicated and the overhead of maintaining mixture of DRAM and Storage Class



(a) Schematic Organization

(b) Organization of Management Area

Figure 1: Organization of Storage Class Memory

Memory address space does not justify its advantage. In this work. we take "fixed binding" approach. Consecutive region in Virtual Memory is bound to Storage Class Memory region. Subsequently, Consecutive entries in the page table are used to address the pages in storage class memory. Fig. 1(a) illustrates the schematic organization of memory subsystem.

To realize Shadow Block mechanism, we need to maintain a certain set of information. We dedicate a certain region of Storage Class Memory to harbor metadata and temporary information required in Shadow Block mechanism. We call this region as Management Area. To be explained in detailed shortly, management area carries the information which is used to guarantee atomicity of Block I/O and to recover the state from unexpected system failure. It should reside in non-volatile region. Fig. 1(b) illustrates this situation.

Storage Class Memory has its own page table. We call it block mapping table. Storage Class Memory page table is responsible for mapping physical page frame number to physical Storage Class Memory block number(Fig. 1(b)). Similar technique is being used in Flash device where Flash Translation Layer(FTL) maps the incoming block number to actual location of the respective block. There exist two type of blocks in Storage Class Memory: *data block* and *shadow block*. Likewise, block mapping table maintains location of *shadow block* and the location of *data block*. The *shadow block* is not visible from page table and is internal to Storage Class Memory.

3.2 Address Translation in Shadow Block

We use block mapping table to locate the destination block and perform out-of-place update. `shadow_block_write` which is write operation of Shadow Block mechanism has two parameters: `shadow_block_write(int block_no, void* buf)`. `block_no` is the logical block number in the Storage Class Memory region and `buf` is the pointer to user buffer. Write operation consults the block mapping table to obtain the physical page frame number for the respective logical block. Page frame number can be obtained from the physical block number by simple computation. Once the page frame number is found, Shadow Block interface uses this value as an index to the page table. At first accesses to a page, referencing page table is essential unless the mapping information is already in TLB. However, after first access to a page, virtual to physical mapping information is hold in TLB. Thus subsequent accesses do not have to reference page table. Because one `shadow_block_write()` consists of several load operations, accessing block mapping table for each load-store operation can be a significant overhead in Shadow Block mechanism. However there exists only one access to block mapping table in our Shadow Block interfaces. This enables us to use Shadow Block mechanism without significant overhead.

1: **procedure** SHADOW_BLOCK_WRITE($block\_no, buf$)
2: $l \leftarrow block\_no$ $/*\ S_1\ */$
3: $tmp \leftarrow M[l]$ $/*\ S_2\ */$
4: MEMCPY($shadow\_block, buf$)
5: $M[l] \leftarrow n_{shadow}$ $/*\ S_3\ */$
6: $n_{shadow} \leftarrow tmp$ $/*\ S_4\ */$
7: $l \leftarrow NULL$ $/*\ S_5\ */$
8: $tmp \leftarrow NULL$ $/*\ S_0\ */$
9: **end procedure**

Figure 3: Write in Shadow Block

3.3 Write Operation in Shadow Block

The `shadow_block_write()` operation makes an update on the *shadow block* instead of performing in-place update. Then, the block mapping table entries for *shadow block* and the block which is to be updated are swapped. To make the Shadow Block 'Atomic', we need to retain history of data as well as index value. To exchange values of two variable, say x and y, we introduce additional variable, say tmp, as a temporary place for holding the value. We perform $[tmp \leftarrow x;\ x \leftarrow y;\ y \leftarrow tmp]$. In Shadow Block mechanism, we first write to *shadow block* and exchange two entries of block mapping table: one for *shadow block* and one for data block. We introduce variable *tmp* to temporarily hold the entry value of the block mapping table. To properly recover the system state after crash, Shadow Block mechanism needs to maintain destination address in persistent manner. Shadow Block copies the destination address to Storage Class Memory so that it can redo after system crash. The variable l resides in Storage Class Memory region and is used to store the destination address of the

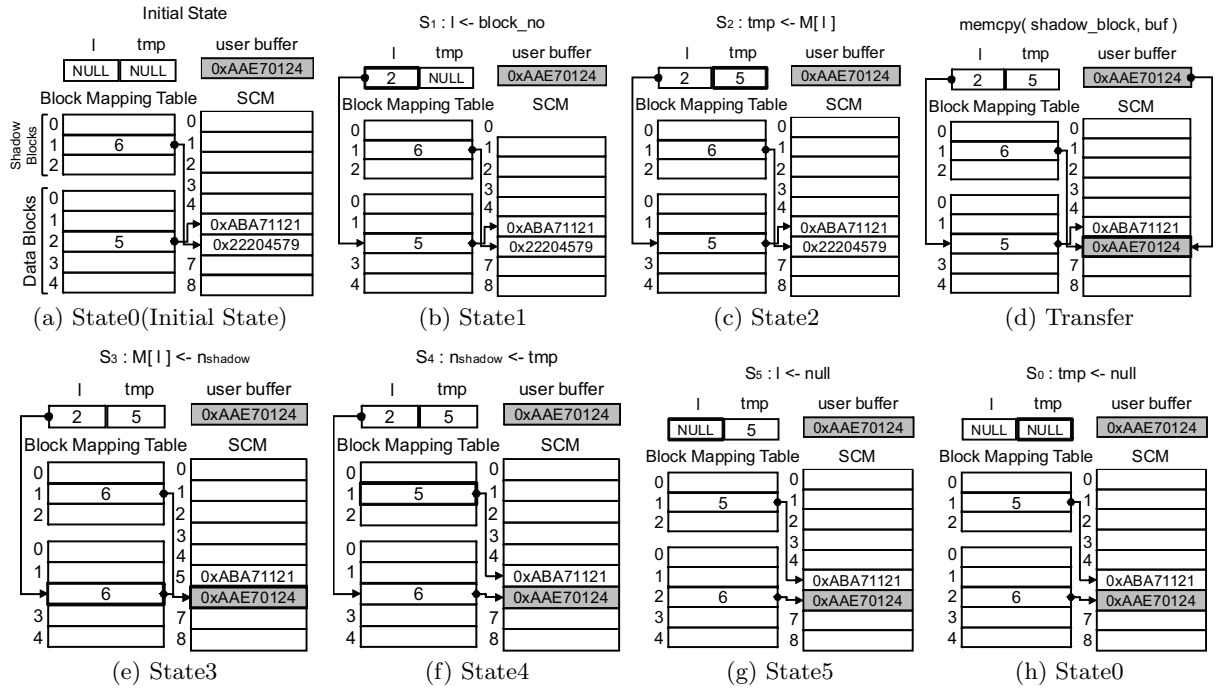


Figure 2: shadow_block_write(2, buf)

`shadow_block_write` operation. Fig. 3 illustrates a pseudo code for `shadow_block_write` operation. In pseudo code, M and n_{shadow} denote the block mapping table and logical block number of *shadow block* respectively. The data transfer is performed into *shadow block* of that physical block is pointed by n_{shadow}.

Let us provide an example(Fig. 2). It writes data to the block 2 in Storage Class Memory region. First, it copies target block number to local variable(Fig. 2(b)). Then, it stores physical block number of logical block 2 to *tmp* variable(Fig. 2(c)). Then, it performs write to *shadow block*(Fig. 2(d)). Next, it exchanges address values in block mapping table(Fig. 2(e), Fig. 2(f)). Finally, it initializes l and *tmp* variables(Fig. 2(g) and Fig. 2(h)).

```
1: if l ≠ NULL & tmp = NULL then
2:     return S1
3: else if l ≠ NULL & tmp ≠ NULL then
4:     if M[l] = tmp then
5:         return S2
6:     else if M[l] = n_shadow then
7:         return S3
8:     else if n_shadow = tmp then
9:         return S4
10:    end if
11: else if l = NULL & temp ≠ NULL then
12:    return S5
13: end if
```

Figure 4: Pseudo Code for Detecting State

3.4 Recovery

state	l	tmp	n_{shadow}	$M[l]$	action
S_0	*null*	*null*	B	C	*none*
S_1	A	*null*	B	C	*undo*
S_2	A	C	B	C	*undo*
S_3	A	C	B	B	*redo*
S_4	A	C	C	B	*redo*
S_5	*null*	C	C	B	*redo*

Table 2: State of each Step of Shadow Block

To recover from partial write, Operating System should be able to identify the state of the write operation when it has failed and should be able to take appropriate action:*redo* or *undo*. In Shadow Block, `shdow_block_write` operation consists of six states, $S_i, i = 0, \ldots, 5$(Fig. 3). Each state can be identified by examining the four variables in management area: l, *tmp*, n_{shadow}, and $M[l]$.

Let us assume that the logical and physical addresses of the destination block of write operation is A and C, respectively and that physical address of the *shadow block* is B. Table 2 illustrates the contents of four variables for each state. Be reminded that these variables are maintained at Storage Class Memory region.

When system restarts after failure, it is possible to identify the state of the write operation at the moment when it has failed, via examining four variables. For example, if both l and *tmp* are $NULL$, system was at S_0. When l and *tmp* is not $NULL$ and $NULL$, respectively, state of write operation was S_1. Fig. 4 illustrates the pseudo code to determine the state of write operation. Based upon the state of the write, we can *redo* the rest of the steps in write operation or *undo* the sequence of the steps performed till the system failure. Henceforth the atomicity is guaranteed.

As described in Fig. 3, write operation updates the *shadow block* after S_2. Therefore, when the final state was S_1, we can *undo* the write simply by resetting l and tmp to $NULL$. When the final state is S_2, it implies that either *shadow block* has not been completely updated or has not reached S_3. We regard both cases as incomplete update and *undo* the write operation by resetting l and tmp. If the system state was S_3, S_4 or S_5, *shadow block* has been successfully updated but the block mapping table has not been properly updated to reflect the shadow page. We *redo* the rest of the steps and completes write operation.

4. PERFORMANCE EVALUATION

4.1 Experiment Setup

We develop prototype I/O subsystem on embedded board. We use 64 MByte SDRAM, 64 Mbit FRAM chip for the main memory and Storage Class Memory layer, respectively. 64 Mbit FRAM chip is the largest scale under current state of art technology. This storage system is built into a SMDK2440 embedded system, which has an ARM 920T microprocessor. FRAM has same access latency as SRAM: 110ns asynchronous read/write cycle time, 4Mb x 16 I/O, and 1.8V operating power. Since the package type of FRAM is 69FBGA(Fine Pitch Ball Grid Array), we develop a daughter board to attach FRAM to the memory extension pin of a SMDK2440 board. The SMDK2440 board supports 8 banks from bank0 to bank7. These banks are directly managed by an Operating System Kernel. We choose bank1(`0x0800_0000`) for FRAM. Shadow Block mechanism is developed on Linux 2.4.20.

We examine two aspects of Shadow Block mechanism: overhead of Shadow Block mechanism and the scalability. First, we compare the I/O performance of legacy `read()`/`write()` and `read()`/`write()` with Shadow Block. Second, we examine the scalability of Shadow Block mechanism. Data structures in management area of the Storage Class Memory need to be protected against concurrent accesses. It is critical that ensuring exclusive access on this data structure does not severely degrade the system performance.

4.2 Atomic vs. Non Atomic I/O

Fig. 5 illustrates the performance of legacy `read()`/`write()` and `read()`/`write()` with Shadow Block mechanism. We measure the bandwidth of sequential I/O. Single thread performs I/O operation. As can be seen, the overhead of Shadow Block is less than 0.1% in `read` and 1% in `write` operation.

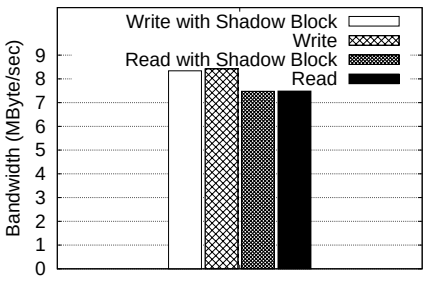


Figure 5: Overhead of Shadow Block Mechanism

4.3 Scalability

To perform Shadow Block write, OS kernel first needs to obtain exclusive lock on management area. A number of threads may issue Shadow Block write operations simultaneously and these operations will be serialized to access management area. We examine if Shadow Block write operation is scalable with a number of threads.

We vary the number of threads and measure the I/O performance. We vary the I/O size to 4, 32, 256, 1024 and 4096 KByte. We implement each thread as POSIX user thread to minimize the overhead of thread switch. Locking the management area is exposed to widely known anomalies, e.g. thundering herd, convoy[14]. Resolving these issues should be dealt with in separate context and we solely focus our effort on efficiency of Shadow Block under varying number of threads. Fig. 6(a) and Fig. 6(b) illustrate the results. Fig. 6(a) illustrates total bandwidth under varying number of threads with different chunk sizes. There is almost no performance degradation until 8 threads. Even if more threads are created, there is no significant throughput reduction. Fig. 6(b) is different manifestation of the same experiment. Here, we observe I/O performance under varying I/O chunk size. When I/O chunk size is larger than 32KByte, the advantage of using larger chunk size becomes insignificant.

5. RELATED WORKS

Imposing block device abstraction on memory address space is not new. Ramdisk[11] and RAMFS[15] use fraction of main memory space as block device. Every I/O operation to Ramdisk therefore entails memory to memory copy. tmpfs resides in Virtual Memory address[13]. Since it resides in VM space, file in tmpfs may be swapped out. These file systems are not designed to store data in persistent manner and contents are reset when power goes down. They do not consider partial write problem. PRAMFS[1] is file system dedicated for persistent RAM. It implement *write* operation as simple `memcpy()` and does not address partial write issue. Jung et. al. proposed Log-Based Block Mapping to address the partial write issue in Storage Class Memory[7]. If storage class memory is used as an I/O device, I/O operation will be very quick. The overhead of switching context and handling interrupt may become overly dominant. Gim et. al.[5] proposed to adaptively switch context in handling I/O request subject to the speed of the underlying block device, workload characteristics and etc. Recently, a number of works proposed to exploit byte-addressable NVRAM as cache or new type of main memory layer mainly to improve the power consumption of legacy DRAM. In all these works, atomic write issue needs to be properly addressed[10, 3, 9]. Kang et. al. examine the performance issue of accessing files in Storage Class Memory[8]. They proposed a file access framework for NVRAM-only computer system which uses single next-generation NVRAM for both primary memory and secondary storage. The proposed framework dynamically allocates memory for both running process and file system. Shadow paging have been proposed to provide atomicity in database transaction. As a means to provide atomicity on arbitrary sized memory object, transactional memory technique has been proposed[12].

6. CONCLUSION AND FUTURE WORK

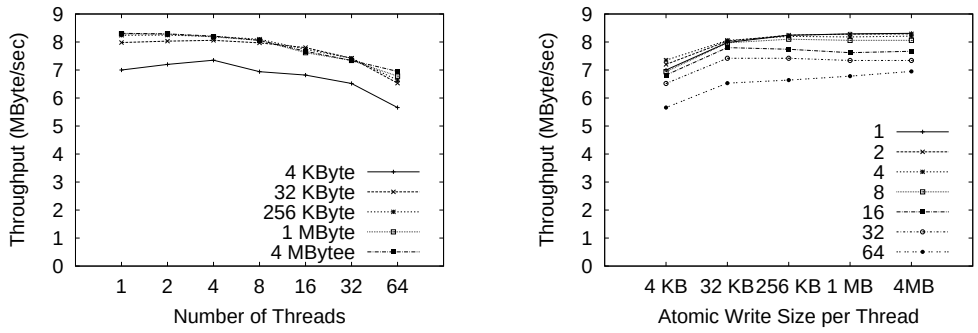


Figure 6: Scalability of Shadow Block

Due to physical characteristics of Storage Class Memory, its advancement and its integration with legacy hierarchical computer organization calls for complete overhauling of modern Operating System. This work pose one key technical issue in using Storage Class Memory as storage: atomicity of I/O operation. Given the speed of Storage Class Memory, it will likely use the existing DRAM interface or use the high speed I/O interface where Storage Class Memory can be accessed in word granularity. To exploit the storage aspect of Storage Class Memory, Operating System should be able to perform I/O in Block unit and "partial write" does not happened. Block I/O on memory address space has long been around and is of no more technical interests in legacy memory and storage device. However, due to access speed, non-volatility and byte-addressability of the Storage Class Memory, this technical issue requires fundamentally different treatment from existing ones. In this work, we examine the various issues involved in providing atomicity of I/O operation in storage class memory. We propose Shadow Block to support atomicity in I/O operation. This mechanism successfully deliver block device abstraction without much overhead(less than 1%). While this approach delivers promising performance result, there still exist a number of technical issues outstanding. They include determining appropriate number of shadow blocks under multi-core CPU, architectural support for Shadow Block and unifying block based access and byte granularity access in the storage class memory region.

7. REFERENCES

[1] Protected and persistent ram filesystem. http://pramfs.sourceforge.net/.

[2] A. L. Brown. Persistent object stores. Technical report, Software Engineering Journal of University of St. Andrews.

[3] R. Freitas and W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4-5):439–448, 2008.

[4] R. Freitas, W. Wilcke, and B. Kurdi. Storage class memory, technology and use. In *Tutorial, 6^{th} USENIX Conference on File and Storage Technologies*, San Jose, CA, USA, 2008.

[5] J. Gim, K. Lee, and Y. Won. Adaptive Context Switch for Very Fast Blcok Device. In *7^{th} USENIX Conferenece on File and Storage Technologies (FAST09)*, San Francisco, CA, USA, Feb 2009. Poster.

[6] K. M. Greenan and E. L. Miller. Prims: making nvram suitable for extremely reliable storage. In *HotDep'07: Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, Berkeley, CA, USA, 2007.

[7] J. Jung, Y. Won, and S. Kang. Supporting Block Device Abstraction on Storage Class Memory. In *Proceedings of Workshop on Computing with Massive and Persistent Data(CMPD08)*. MSST, 2008.

[8] S. Kang, S. Park, and Y. Won. Accessing Files in Storage Class Memory-based Computer Systems. In *Proceedings of Workshop on Computing with Massive and Persistent Data(CMPD08)*. MSST, 2008.

[9] K. Kant. Exploiting NVRAM for Building Multi-Level Memory Systems. In *International Workshop on Operating System Technologies for Large Scale NVRAM(NVRAMOS08)*, Jeju, Korea, Oct 2008. http://dcslab.hanyang.ac.kr/nvramos08/KrishnaKant.pdf.

[10] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proceedings of The 36th International Symposium on Computer Architecture*, Austin, Texas, USA, June 2009. ISCA.

[11] N. M. How to use a ramdisk for linux. http://www.linuxfocus.org/English/November1999/article124.html, 2003.

[12] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[13] P. Snyder. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 EUUG Conference*, pages 241–248.

[14] U. Vahalia. *UNIX internals: the new frontiers.* Prentice Hall Press Upper Saddle River, NJ, USA, 1995.

[15] J. Warnier. About ramdisks and ramfs. http://plume.bxlug.be/articles/7.

[16] T. Ylönen. Concurrent Shadow Paging: A New Direction for Database Research. Technical report, Laboratory of Information Processing Science, 1992.

[17] W. Zhao, E. Belhaire, Q. Mistral, E. Nicolle, T. Devolder, and C. Chappert. Integration of Spin-RAM technology in FPGA circuits. In *Solid-State and Integrated Circuit Technology, 2006. ICSICT'06. 8th International Conference on*, pages 799–802, 2006.