Deterministic Crash Recovery for NAND Flash Based Storage Systems

Chi Zhang[†], Yi Wang[†], Tianzheng Wang[§], Renhai Chen[†], Duo Liu[‡], Zili Shao[†]
[†] Department of Computing, The Hong Kong Polytechnic University
[§]Department of Computer Science, University of Toronto
[‡]College of Computer Science, Chongqing University

ABSTRACT

NAND flash memory has long been the dominant storage medium in mobile devices. However, power failure may occur at any time and result in loss of important data. Crash recovery therefore becomes vitally important in NAND flash memory storage systems. As flash translation layer (FTL) directly manages flash memory using various metadata, the problem of FTL crash recovery in NAND flash is how to efficiently and effectively maintain and recover the consistency of FTL metadata after system crash.

In this paper, we present DCR, a deterministic approach to crash recovery for NAND flash based storage systems. The basic idea is to exploit the determinism of FTL and reproduce events that happened between the last checkpoint and the crash point during crash recovery. Different from existing approaches which have to scan the whole flash memory chip, we show that DCR can recover the system more efficiently by only checking a limited number of blocks based on deterministic FTL operations. We have implemented DCR for a block-level FTL and compared it with a popular version-based scheme using an ARM11-based embedded evaluation board. Experimental results show that DCR can greatly reduce recovery time and guarantee the consistency of FTL metadata after recovery.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Secondary Storage*; B.3.4 [**Memory Structures**]: Reliability, Testing, and Fault-Tolerance—*Error-checking*

General Terms

Design, Experimentation, Performance, Reliability

Keywords

NAND flash memory, reliability, crash recovery

1. INTRODUCTION

DAC '14, June 01 - 05 2014, San Francisco, CA, USA.

NAND flash memory is widely used in embedded systems, such as smartphones and tablets. With more and more functionalities being integrated, mobile devices now often suffer from sudden power failure because of operating system crash and the limited battery capacity. Embedded systems running in harsh environments may also have the same problem. If a flash memory page is storing important information (e.g., file system metadata, address mappings), the data corruption of the page is very serious, as it may cause an unintended change in functionality of the entire system [14]. Crash recovery therefore becomes an important component in these systems.

Different from hard disks, the "erase-before-write" limitation of flash memory requires a special layer of system software called flash translation layer (FTL) to emulate a block device interface for backward compatibility [4, 18]. File systems and applications can use flash memory as if they were using a hard disk. To manage flash memory, an FTL uses various metadata (such as address mapping tables), which directly determine whether data can be stored and accessed correctly. Crash recovery module must recover the system to a consistent state by correctly manipulating FTL metadata.

This paper focuses on recovering FTL metadata rather than upper level file systems metadata. Crash recovery has been extensively studied in file systems. Modern file systems such as Ext4 often keep a journal to recover from power failure. However, used on top of an FTL, they rely on the correctness of the underlying storage system to correctly function and recover consistent states. Errors in FTL metadata can result in disastrous consequences in file system recovery. Since crash recovery techniques in modern file systems normally target at hard disks, these techniques cannot be applied to solve crash recovery problems in flash memory. Therefore, designing a correct and robust crash recovery mechanism for FTL not only affects the integrity of data in flash memory but also influences the functionality of the entire system.

Most existing approaches for FTL metadata crash recovery focus on improving recovery efficiency by avoiding scanning all pages in a flash memory block. In flash-specific file systems, version-based schemes [2, 15] are used to identify updated blocks and accelerate crash recovery. File-system-aware FTLs use metadata filter and summary page in each block to reduce overhead of crash recovery [17]. There are also schemes targeting on specific types of FTLs [8, 12, 13]. For example, some previous studies [16, 19] showed that in segment-based FTL, a superblock is helpful in reducing the

number of blocks scanned. Despite the effectiveness of existing approaches, they suffer from sub-optimal recovery speed as the recovery algorithm has to touch all blocks. In the past decade, flash memory has witnessed a nearly 100x increase of capacity [6]. As the trend continues, it becomes very important to improve recovery speed after system crash.

This paper for the first time presents a deterministic approach to crash recovery in NAND flash storage systems. The basic idea is that events occurred between the last checkpoint and crash point are limited and deterministic for a particular FTL; thus, crash recovery can be more easily done if we can reproduce what occurred from a recovery checkpoint to the crash point. Based on this idea, we develop a deterministic crash recovery scheme called DCR for a representative block-level FTL that is widely used in embedded systems. Different from previous work which requires that all blocks be checked, we show that DCR only needs to check a limited number of blocks. Time required for crash recovery is therefore reduced.

We have implemented and evaluated DCR against a popular version-based crash recovery scheme [15] using a real hardware platform. The platform employs an ARM 11 processor core with ARMv6 architecture. Experimental results show that, compared with the version-based crash recovery scheme, DCR can achieve an average reduction of 20.68% on crash recovery time for real applications. For benchmark Postmark with the capacity of 128 GB, DCR can recover FTL metadata within 13.34 seconds, while the version-based scheme will take 62.79 seconds.

The rest of this paper is organized as follows. Section 2 gives the background and motivation of crash recovery in flash memory based storage systems. Section 3 presents details on DCR. In Section 4, we present experimental results. Finally, Section 5 provides the conclusion of this paper.

2. BACKGROUND AND MOTIVATION

2.1 NAND Flash Memory Storage Systems

In NAND flash memory storage systems, the unit of a read/write operation is a page and the unit of an erase operation is a block. Flash translation layer is normally adopted to conceal the unfavorable properties, i.e., "out of place update", in flash memory storage systems. As shown in Fig. 1(a), the FTL emulates a block device interface for traditional file systems and applications. It consists of wear leveler, address translator, and garbage collector. The wear-leveler distributes write or erase operations evenly across all flash memory blocks [9]. The address translator provides logical to physical address translations by utilizing a mapping table. The garbage collector is responsible for reclaiming space for incoming requests by erasing obsolete blocks.

2.2 FTL Mapping Strategy

There are mainly three types of FTL according to the mapping granularity used: page-level, block-level, and hybrid-level. In this paper, we implement DCR on top of a block-level FTL as it is widely used in various devices. In the block-level FTL we adopted, each logical block is mapped to a physical primary block, in which physical page offsets are identical to logical page offsets in the logical block. A replacement block is used to accommodate updates to the primary block, and free pages in it are allocated sequentially. Though DCR is implemented on top of a block-level FTL, it can also be applied to other types of FTL because the page allocation strategy is reproducible and deterministic in most FTLs.

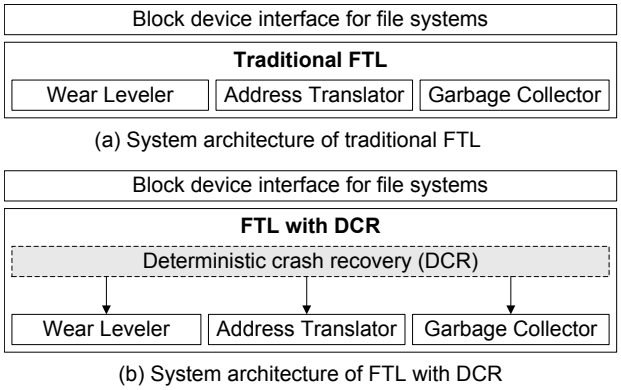

Figure 1: Comparison between (a) A traditional FTL and (b) FTL with DCR.

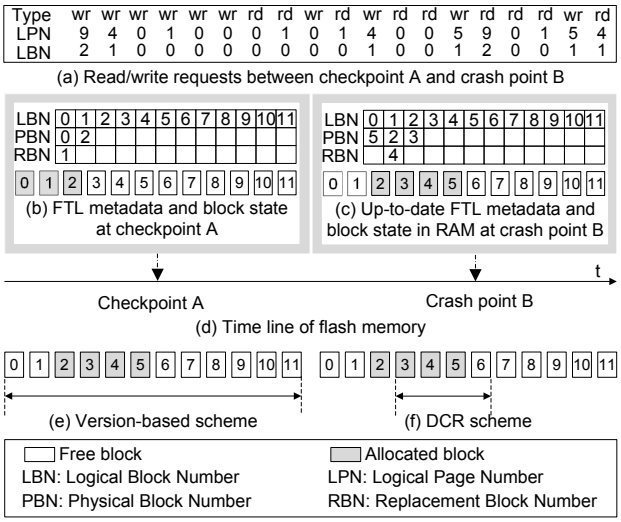

Figure 2: A motivational example.

2.3 Motivational Example

FTL metadata includes block mapping table and other related state information such as block erase count. All upper layer applications and file systems rely on correct FTL metadata to find the correct data in flash memory. During normal processing, FTL metadata is cached in RAM and flushed back to a reserved area in flash memory periodically. When there is a power failure, FTL metadata and data in flash memory will become inconsistent. If FTL metadata cannot be recovered correctly, mapping information will become inaccurate and the upper level file systems will not be able to recover and function correctly.

Fig. 2 shows a motivational example. We assume that there are 12 blocks in flash memory and only 3 blocks are updated from the most recent checkpoint to crash point. The figure shows the process of recovering FTL metadata for these 12 blocks. Fig. 2(a) shows the I/O requests issued to the FTL by the file system. The FTL mapping table at the last checkpoint is shown in Fig. 2(b), with three allocated

physical blocks (blocks 0, 1 and 2). The FTL mapping table in Fig. 2(b) also represents the case before crash recovery, since updated FTL metadata in RAM has lost with power failure. After serving the requests in Fig. 2(a), the mapping table is transformed and shown in Fig. 2(c), where blocks 3, 4 and 5 are allocated and blocks 0 and 1 are freed.

We assume that the system is crashed before the mapping table in Fig. 2(b) is made durable. The system restarts from the last checkpoint before the system crash, which is shown in Fig. 2(b). It recorded the block states (either free or allocated) by the time the mappings shown in Fig. 2(b) was made durable. Only blocks 0, 1 and 2 were allocated. As shown in Fig. 2(e), at system restart, a traditional version-based scheme will have to scan all the blocks to figure out block states and set up the mappings. Using the last checkpoint in Fig. 2(b) and following the FTL block allocation strategy, as shown in Fig. 2(f), we start to scan from block 3 (block 3 is the first empty block based on allocation strategy) and stop at block 6 because it is empty (blocks after block 6 are also empty and unused). Only four blocks have to be scanned. By inferring the FTL allocation strategy and using the last checkpoint, we can achieve the same correct result while only scanning a limited number of blocks and reducing crash recovery time.

3. DCR: DETERMINISTIC CRASH RECOVERY SCHEME FOR NAND FLASH MEMORY

This section presents DCR, a Deterministic Crash Recovery scheme for NAND flash memory storage systems. DCR process is presented in Fig. 3. As shown in Fig. 3(a), DCR handles three typical data crash cases, including regular case (in phase 1), special garbage collection (in phase 2), and wear leveling (in phase 3). In the following sections, we present the detailed procedures for these three crash recovery cases.

3.1 Crash Recovery for Three Typical Cases

3.1.1 Crash Recovery for Regular Case

In FTL-based NAND flash memory storage systems, FTL normally issues the basic operations, such as read, write, and erase operations to the underlining Memory Technology Device (MTD) layer and NAND flash memory chip. As the first phase of DCR, we handle the data crash for these regular cases. For each data block after the checkpoint, DCR will perform update function if the data block is not an empty block. The update function is presented in Fig. 3(b). First of all, DCR needs to check the block that may be updated after the checkpoint. When it reads the spare area of each page, it can find the logical page number and calculate logical block number (LBN). If there is no valid page in this block, we can erase this block since no valid page exists. After LBN is calculated, we can find the victim primary block number (VPN) and victim replacement block number (VRN) based on the current mapping table.

There are three conditions for the checking outcome of VPN and VRN. The first condition is that there is no VPN in the mapping table. The second condition is that the mapping table has VPN but no VRN for the logical block number. Both cases are handled by the function *UpdateMetadataAllocateNew(LBN, PBN, VPN)*. The last condition is that both VPN and VRN are in the mapping table. For

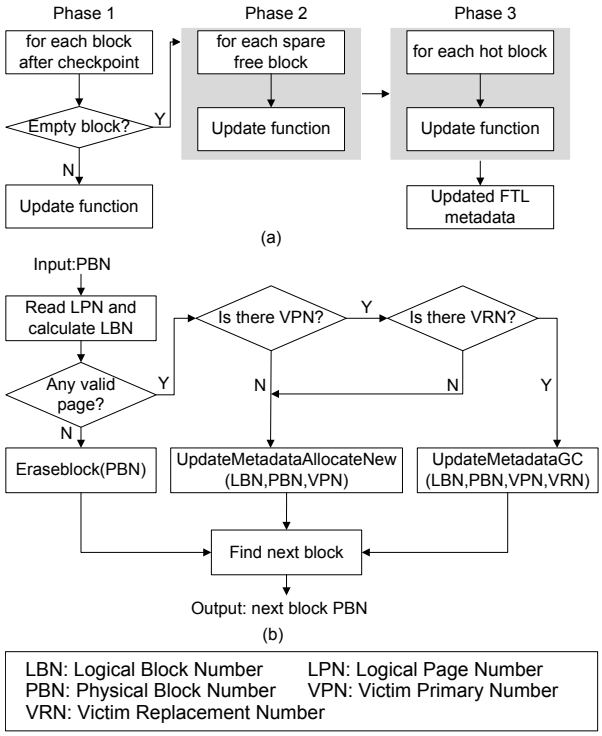

Figure 3: (a) DCR performs crash recovery in three phases. (b) The update function.

this condition, we present *UpdateMetadataGC(LBN, PBN, VPN, VRN)* to deal with this condition. After updating FTL metadata, we need to find the next block that may be allocated. The block can be found based on the allocation strategy. If the block found based on the allocation strategy is empty, we continue checking in the next phase.

3.1.2 Crash Recovery for Special Garbage Collection

Crash recovery for special garbage collection handles one special case in which the crash may occur during the garbage collection. In special garbage collection, a write operation to the same logical block address may trigger multiple times of garbage collection operations, and an empty block may be updated after the checkpoint and erased to be an empty block again before the crash point.

To solve this problem, DCR creates a list called spare free block list to reserve a small number of physical block numbers. When the special garbage collection happens, the physical blocks in spare free block list are allocated. In Fig. 3(a), during crash recovery, DCR checks blocks in the list and it will continue until the end of the spare free block list is reached. The recovery procedure for special garbage collection is the same as the one for regular case.

3.1.3 Crash Recovery for Wear-Leveling

In NAND flash memory storage systems, wear-leveling operation is used to ensure that erase operations are evenly distributed across the flash memory chip and physical blocks can achieve similar block erase counts. Crash may happen during the wear-leveling, which may affect the data integrity of flash.

Crash recovery for wear-leveling handles this case, and DCR uses a hot block list to predict the physical updated blocks. The hot block list reserves a small number of physical block numbers. When system crash happens during wear-leveling, DCR will allocate a physical block in hot block list. As shown in Fig. 3(a), during crash recovery, DCR checks physical blocks in the hot block list and will continue until the end of the hot block list is reached. The recovery process has some minor differences with the regular case, since wear leveling only needs to consider one victim block rather than two blocks of the garbage collection in regular case.

3.2 Crash Recovery Schemes

This section presents the detailed crash recovery schemes. In DCR, we use two functions, i.e., *UpdateMetadataAllocateNew()* and *UpdateMetadataGC()*, to reproduce the potential affected physical blocks and reduce the time cost for crash recovery. *UpdateMetadataAllocateNew()* is presented in Algorithms 3.1.

UpdateMetadataAllocateNew() is a function that handles the case of allocating a new physical block to a logical block. The inputs of this function are logical block number (LBN), physical block number (PBN), and victim primary block number (VPN). The output is block control block (*p_bcb*), which is the control block of PBN. *v_bcb* and *entry* are the control block of VPN and the mapping entry of LBN, respectively. If *p_bcb* represents a primary block, we need to check whether the mapping entry has one primary block VPN. If VPN exists, we update the block state of VPN to $FREE$ and add the number of erase times by 1. In addition, we update the primary block of mapping entry. If *p_bcb* represents a replacement block, we update the replacement block. We need to update block state of *p_bcb* to $ALLOCATED$ when *p_bcb* is a primary block or a replacement block.

UpdateMetadataGC() is a function that handles the case of garbage collection crash. As shown in Algorithm 3.2, the inputs of this function are logical block number (LBN), physical block number (PBN), victim primary block number (VPN), and victim replacement block number (VRN). Output is block control block (*p_bcb*), which is the control block of PBN. *v_bcb*, *r_bcb* and *entry* are the control block of VPN, VRN and the mapping entry of LBN, respectively.

If *p_bcb* represents a primary block, we need to check whether the mapping entry has primary block VPN and replacement block VRN. If *v_bcb* is an empty block, we update the block state of VPN to $FREE$, update the block state of PBN to $ALLOCATED$, add the number of erase times by 1, and update the primary block of mapping entry. If *r_bcb* is an empty block, we update the block state of VRN to $FREE$ and add the number of erase times by 1. If *r_bcb* is not empty, we erase the block *r_bcb*. Besides, If *v_bcb* is not erased, we erase the block *p_bcb*. Otherwise, we will erase the blocks *v_bcb* and *r_bcb*. Finally, we update the primary block of the mapping entry and update the block state of PBN to $ALLOCATED$.

If *p_bcb* represents a replacement block, the function will update the block state of VPN to $FREE$, update the block state of PBN to $ALLOCATED$, add the number of erase times by 1, and update the primary block of mapping entry to NULL and replacement block to PBN. In addition, *v_bcb* and *r_bcb* block states need to be changed to $FREE$, and the number of erase times needs to be added by 1.

Algorithm 3.1 $UpdateMetadataAllocateNew$

Input: Logical block number (LBN), physical block number (PBN), victim primary block number (VPN).
Output: *p_bcb*: block control block.

Let $p\_bcb = get\_block\_cb(PBN)$.
Let $v\_bcb = get\_block\_cb(VPN)$.
Let $entry = get\_mapping\_entry(LBN)$.
if *p_bcb* represents a primary block **then**
  if $VPN \neq \emptyset$ **then**
    $v\_bcb.state \leftarrow FREE$.
    $v\_bcb.erasetimes \leftarrow v\_bcb.erasetimes + 1$.
  end if
  $entry.primary\_blk \leftarrow p\_bcb$.
else
  $entry.replacement\_blk \leftarrow p\_bcb$.
end if
$p\_bcb.state \leftarrow ALLOCATED$.
return *p_bcb*.

Algorithm 3.2 $UpdateMetadataGC$

Input: Logical block number (LBN), physical block number (PBN), victim primary block number (VPN), victim replacement block number (VRN).
Output: *p_bcb*: block control block.

Let $p\_bcb = get\_block\_cb(PBN)$.
Let $v\_bcb = get\_block\_cb(VPN)$.
Let $r\_bcb = get\_block\_cb(VRN)$.
Let $entry = get\_mapping\_entry(LBN)$.
if *p_bcb* represents a primary block **then**
  if *v_bcb* is an empty block **then**
    $v\_bcb.state \leftarrow FREE$.
    $v\_bcb.erasetimes \leftarrow v\_bcb.erasetimes + 1$.
    $entry.primary\_blk \leftarrow p\_bcb$.
    $p\_bcb.state \leftarrow ALLOCATED$.
    if *r_bcb* is an empty block **then**
      $r\_bcb.state \leftarrow FREE$.
      $r\_bcb.erasetimes \leftarrow r\_bcb.erasetimes + 1$.
    else
      Erase block(VRN).
    end if
  else
    if no error in *v_bcb* **then**
      Erase block(PBN).
    else
      Erase block(VPN), Erase block(VRN).
      $entry.primary\_blk \leftarrow p\_bcb$.
      $p\_bcb.state \leftarrow ALLOCATED$.
    end if
  end if
else
  $entry.replacement\_blk \leftarrow p\_bcb$.
  $entry.primary\_blk \leftarrow$ NULL.
  $p\_bcb.state \leftarrow ALLOCATED$.
  $v\_bcb.state \leftarrow FREE$.
  $r\_bcb.state \leftarrow FREE$.
  $v\_bcb.erasetimes \leftarrow v\_bcb.erasetimes + 1$.
  $r\_bcb.erasetimes \leftarrow r\_bcb.erasetimes + 1$.
end if
return *p_bcb*.

4. EXPERIMENTS

4.1 Experimental Setup

The objective of the evaluation is to quantify the gains of DCR over the conventional version-based approach [15] in terms of recovery time. Both DCR and version-based approach have been implemented in a real hardware platform.

Table 1: The comparison for recovery time of our DCR approach and the version-based approach [15] with benchmarks *Mplayer*, *File-copy1*, *File-copy2*, and *File-copy3*.

Benchmarks	Size (MB)	# of updated blocks	DCR (sec)				Version-based (sec)	Time reduction
			Experiment1	Experiment2	Experiment3	Average		
Mplayer	8.900	0	0.335	0.335	0.335	0.335	0.473	29.10%
File-copy1	1.024	8	0.575	0.576	0.575	0.575	0.748	23.13%
File-copy2	0.256	2	0.403	0.398	0.400	0.400	0.542	26.13%
File-copy3	10.752	84	3.222	3.220	3.219	3.220	3.367	4.35%

Table 2: The comparison for recovery time of our DCR approach and the version-based approach [15] with benchmarks *Bonnie*, *Postmark*, and *Tiotest*.

Benchmarks	Crash	Size (MB)	# of updated blocks	DCR (sec)				Version-based (sec)	Time reduction
				Experiment1	Experiment2	Experiment3	Average		
Bonnie	25%	6.144	48	1.849	1.851	1.850	1.850	2.127	13.00%
	50%	12.160	95	3.494	3.495	3.494	3.495	3.745	6.68%
	75%	18.048	141	4.833	4.835	4.836	4.835	5.030	3.88%
	100%	24.064	188	6.749	6.750	6.750	6.750	6.950	2.88%
Postmark	25%	2.048	16	0.861	0.862	0.862	0.862	1.024	15.87%
	50%	4.352	34	1.431	1.430	1.429	1.430	1.644	13.02%
	75%	6.400	50	1.997	1.995	1.994	1.995	2.195	9.11%
	100%	8.576	67	2.632	2.633	2.364	2.633	2.781	5.32%
Tiotest	25%	20.736	162	5.840	5.841	5.840	5.840	6.055	3.55%
	50%	41.216	322	11.411	11.408	11.410	11.410	11.567	1.35%
	75%	61.952	484	16.964	16.965	16.965	16.965	17.150	1.08%
	100%	82.432	644	22.520	22.520	22.520	22.520	22.660	0.62%

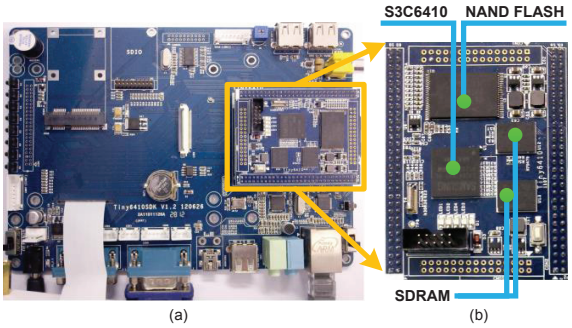

Figure 4: Hardware evaluation platform with an ARM 11 processor core and a 1 GB NAND flash memory.

Fig. 4 illustrates the embedded evaluation board. The board employs an ARM 11 processor core (Samsung S3C6410) with ARMv6 architecture. The ARM processor core is running at 532MHz, and it consists of a 16 KB instruction cache and a 16 KB data cache. The platform has a core board with a 1 GB NAND flash memory and 256 MB SDRAM, and a mother board with some physical interfaces, such as RJ45 interface. One pin connector is used to connect the core board and the mother board.

In the experiments, we adopt Linux kernel 2.6.38 as operating system, and Ext2 as file system. The block size, page size, OOB size for each page are 128 KBytes, 2 KBytes, 64 Bytes, respectively. The number of pages in a physical block of the NAND flash memory chip is 64.

We conduct experiments using both real applications on the evaluation board and standard benchmarks which are commonly used in the research community. Specifically, we run a video program and perform three file copy operations on the evaluation board. These benchmarks are named as *Mplayer*, *File-Copy1*, *File-Copy2*, *File-Copy3*, respectively. They can reflect the real workload of the system in accessing the NAND flash memory chip. We also use standard benchmarks *Bonnie* [3], *Postmark* [7], and *Tiotest* [1] to evaluate the proposed approach.

DCR is implemented based on the block-level address mapping. We have implemented DCR with several fundamental functions of FTL, including read, write, garbage collection, wear-leveling and the management of FTL metadata. In DCR, we need to store two lists, i.e., spare free block list and hot block list. They are also implemented as a component of an FTL to accomplish the deterministic crash recovery.

4.2 Results and Discussion

In this section, we present the experimental results of DCR and the version-based approach. The version-based scheme utilizes the time version to distinguish the updated block and reduces the time for crash recovery. We compare our approach with the version-based approach because of its reasonably good performance for crash recovery. To make fair comparisons, we run each benchmark for three times and compare the results with the version-based approach.

The experimental results are presented in Tables 1 and 2, respectively. In these tables, *size* represents the size or capacity of each benchmark; *# of updated blocks* represents the number of physical blocks that are programmed before crash recovery. In Table 2, *crash* represents the point we shut down the power supply of the evaluation board. This percentage denotes that the number of physical blocks that have been programmed divided by the total number of physical blocks that should be programmed for a benchmark.

Table 1 shows experimental results of four real applications running on the evaluation board. Compared with the version-based scheme, DCR can achieve an average reduction of 20.68% for *File-copy* benchmarks. For benchmark *Mplayer*, DCR can obtain almost 30% of time reduction. Table 2 presents experimental results for benchmarks *Bonnie*, *Postmark*, and *Tiotest*. From the experimental results, compared with the version-based approach, DCR can achieve average reductions of 6.61% for benchmark *Bonnie*, 10.83% for benchmark *Postmark*, and 1.65% for benchmark *Tiotest*. From the experimental results, we also find that, the time reduction is closely related to the number of updated blocks. The percentage is large with small amount of updated blocks, since version-based scheme needs to scan all the blocks of NAND flash.

Our current implementation of DCR only relies on two lists at worst case, which preserve about 1% of all the blocks. If the sizes of these lists decrease, the time reduction percentage is even larger and the advantage of DCR methodology will be much more obvious. Besides, all the discussions rely on a fixed size of NAND flash. Comparing two different methods, we estimate the crash recovery time of benchmark Postmark with increasing capacity of 4 GB, 8 GB, 16 GB, 32 GB, 64 GB and 128 GB. For benchmark Postmark with the capacity of 64 GB, DCR can recover FTL metadata within 7.9 seconds, compared with 32.55 seconds based on version-based scheme. DCR crash recovery time increases from 2.8 seconds with the capacity of 4 GB to 13.34 seconds with the capacity of 128 GB, but version-based scheme crash recovery time increases from 4.20 seconds with 4 GB to 62.79 seconds. With the larger flash memory, the reduction of crash recovery time is much larger. For this case, a version-based scheme may increase the time of crash recovery, since it needs to scan all blocks, which may affect the performance of crash recovery severely.

5. CONCLUSION

This paper presents DCR, a deterministic approach to crash recovery for NAND flash based storage systems. DCR utilizes the determinism of FTL and reproduces the physical blocks that need to be checked during crash recovery. The proposed method is implemented on a real hardware platform. The experimental results have shown that DCR outperforms a popular version-based scheme in terms of recovery time and runtime overhead. In the future, we plan to extend DCR to support systems with emerging memory techniques, e.g., PRAM, memristor [5, 11], STT-RAM [10, 20]. Using the hybrid memory architecture may further enhance the reliability of flash memory storage system and improve the efficiency of crash recovery. Making upper layer applications such as file systems aware of underlying recovery mechanisms is also a promising direction for effective crash recovery.

6. ACKNOWLEDGEMENT

The work described in this paper is partially supported by the grants from the Germany/Hong Kong Joint Research Scheme sponsored by the Research Grants Council of Hong Kong and the Germany Academic Exchange Service of Germany (Reference No.G_HK021/12), National Natural Science Foundation of China (Project 61272103, 61373049 and 61309004), National 863 Program 2013AA013202, Research Fund for the Doctoral Program of Higher Education of China (20130191120030), Chongqing cstc2012ggC40005 and cstc2013 jcyjA40025, and the Hong Kong Polytechnic University (4-ZZD7,G-YK24, G-YM10 and G-YN36).

7. REFERENCES

[1] Tiotest Benchmark. *http://linux.die.net/man/1/tiotest*.
[2] Aleph One Ltd. Yaffs 2 Specification. *http://www.yaffs.net/yaffs-2-specification*.
[3] T. Bray. The bonnie benchmark. *http://www.textuality.com/bonnie*, 1996.
[4] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of Flash Translation Layer. *J. Syst. Archit.*, 55(5-6):332–343, May 2009.
[5] M. Hu, H. Li, Y. Chen, X. Wang, and R. E. Pino. Geometry variations analysis of tio 2 thin-film and spintronic memristors. In *ASP-DAC '11*, pages 25–30, 2011.
[6] ISSCC Press. ISSCC 2013 tech trends. In *ISSCC '13*, 2013.
[7] J. Katcher. Postmark: A new file system benchmark. 1997.
[8] C. Lee and S.-H. Lim. Efficient Logging of Metadata Using NVRAM for NAND Flash based File System. In *IEEE Transaction on Consumer Electronics*, volume 58, 2012.
[9] D. Liu, Y. Wang, Z. Qin, Z. Shao, and Y. Guan. A Space Reuse Strategy for Flash Translation Layers in SLC NAND Flash Memory Storage Systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(6):1094–1107, 2012.
[10] A. K. Mishra, X. Dong, G. Sun, Y. Xie, N. Vijaykrishnan, and C. R. Das. Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs. In *ISCA '11*, pages 211–216, 2011.
[11] D. Niu, Y. Chen, C. Xu, and Y. Xie. Impact of process variations on emerging memristor. In *DAC '10*, pages 877–882, 2010.
[12] S. S. Rizvi and T.-S. Chung. JAM: justifiable allocation of memory with efficient mounting and fast crash recovery for NAND flash memory file systems. *Int. Arab J. Inf. Technol.*, 7(4):395–402, 2010.
[13] L. Shi, J. Li, C. J. Xue, and X. Zhou. Virtual Memory and Write Buffer Management for Flash-based Storage Systems. In *IEEE Transactions on Very Large Scale Integration Systems(TVLSI*, volume 21, pages 706–719, Apr. 2013.
[14] Y. Wang, L. Bathen, N. Dutt, and Z. Shao. Meta-Cure: A reliability enhancement strategy for metadata in NAND flash memory storage systems. In *2012 49th ACM/EDAC/IEEE Design Automation Conference (DAC),*, pages 214–219, June 2012.
[15] C.-H. Wu, T.-W. Kuo, and L.-P. Chang. The design of efficient initialization and crash recovery for log-based file systems over flash memory. *Trans. Storage*, 2(4):449–467, Nov. 2006.
[16] C.-H. Wu and H.-H. Lin. Timing Analysis of System Initialization and Crash Recovery for a Segment-Based Flash Translation Layer. *ACM Trans. Des. Autom. Electron. Syst.*, 17(2):14:1–14:21, Apr. 2012.
[17] P.-L. Wu, Y.-H. Chang, and T.-W. Kuo. A file-system-aware FTL design for flash-memory storage systems. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 393–398, 2009.
[18] C. Xue, Y. Zhang, Y. Chen, G. Sun, J. Yang, and H. Li. Emerging non-volatile memories: opportunities and challenges. In *Hardware/Software Codesign and System Synthesis(CODES+ISSS)*, 2011.
[19] K. S. Yim, J. Kin, and K. Koh. A fast start-up technique for flash memory based computing systems. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 843–849, 2005.
[20] Y. Zhang, L. Zhang, W. Wen, G. Sun, and Y. Chen. Multi-level cell STT-RAM: Is it realistic or just a dream? In *ICCAD '12*, pages 526–532, 2012.