JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 27, 805-832 (2011)

Backup Metadata As Data: DPC-tolerance to Commodity File System

YOUNG JIN YU, DONG IN SHIN, HYEONG SEOG KIM,
HYEONSANG EOM AND HEON YOUNG YEOM
School of Computer Science and Engineering
Seoul National University
599 Gwanak-ro, Gwanak-gu, Seoul, Republic of Korea

Backup Metadata As Data (MAD) is a user-level solution that enables commodity file systems to replicate their critical metadata and to recover from disk pointer corruptions. More specifically, it extracts disk pointers from file system and saves them as user data. When some data blocks become inaccessible due to pointer corruptions, Backup MAD restores access paths to them either by copying the blocks to another file system or by directly updating on-disk structures of file system. The latter technique helps Backup MAD restore lost files faster than any other recovery solution because data blocks are not moved during restoration. Also, as the technique relies on disk pointers extracted from a consistent file system state, it can rescue up to 50% more files than a scan-based recovery tool that infers block dependencies from a corrupted partition. We demonstrate the effectiveness of our technique by two real implementations, MAD-NTFS and MAD-ext2. Backup MAD enhances dependability of file system by protecting disk pointers on behalf of file system.

Keywords: file system, reliability, disk pointer corruption, disk pointer types, disk pointer snapshot, in-partition restoration, out-of-partition restoration, OS-aware recovery, ordered recovery

1. INTRODUCTION

Data blocks of a file share their *liveness* with metadata blocks. The block dependency is represented by *disk pointer* which file system heavily relies on for many reasons; it may be used as the location of user data to access the data. If disk pointers are corrupted, file system would lose a way to track user data even though the data still remains uncorrupted.

Disk pointer corruption (DPC) problem has come into real life [1, 2, 3], and lowered the dependability of file system. For example, any bit stored in a storage device can be lost due to *software bugs* or *silent corruptions* [4, 5, 6, 7, 8]; if the corrupted bit is a part of an important disk pointer, critical data loss would occur. In a broad sense, intentionally or unintentionally modifying disk pointers can be regarded as DPC; a certain type of viruses infects disk pointers in a boot block, and human error, e.g. accidental formatting, initializes metadata region containing disk pointers [9, 10].

Recognizing the seriousness of DPC, numerous solutions have attempted to find

* This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2010-(C1090-1011-0004)). Hyeonsang Eom is the corresponding author of this paper.

ways to prevent permanent data loss caused by this type of corruption. They involve *metadata replication* [11, 12], *integrity checker* [13, 14], *disk scan-based recovery tool* [15, 16], and *data backup*. Each of them makes its own way to recover data by replicating, repairing, reasoning or re-assigning disk pointers inside file system, but has several weaknesses in terms of correctness, restoration time, or storage cost.

In this paper, we propose *Backup MAD* architecture, a disk-pointer recovery framework to complement the previous approaches. Backup MAD ensures *DPC-tolerance* to commodity file systems by protecting disk pointers on behalf of file system, decoupling the fates of data and metadata blocks. DPC-tolerant file systems prevent any fault from propagating outside the corrupted region. DPC-tolerant file systems are thus more dependable than those not having the feature.

Backup MAD is very correct in the sense that it can safely restore all of lost files as long as their data blocks are kept uncorrupted. Although an integrity checker or a scan-based recovery tool can repair certain inconsistency caused by DPC, its restoration procedure is usually slow [17] and sometimes unsafe [18], incurring further data loss or violation of file system policy. For instance, e2fsck suffers from *false parenthood problem* [18]. Both of the solutions are inherently vulnerable to the case where a partition gets so corrupted that critical disk pointers can't be inferred from it, which leads to significant fault propagation. In contrast, Backup MAD relies on a snapshot image extracted from a consistent file system state and restores lost data safely based on the correct information.

Backup MAD is space- and time- efficient since it backs up disk pointers only and hardly moves data blocks during restoration. Metadata replication technique achieves the same goals by replicating only critical metadata across a few regions and restoring valid one from them. However, the approach is tightly coupled with OS kernel and usually puts replication operations into a critical path [19]. On the other hand, Backup MAD is a user-level solution, decoupled from file system. It doesn't incur overhead for ensuring fault tolerance and eases deployment by not requiring OS to be modified or to have any specialized device driver.

The main contribution of this paper is that we 1) classify common types of disk pointers of file system to help a user application understand complex on-disk format, and 2) devise a user-level solution to restore important disk pointers inside file system by balanced usage between file system API and raw access. Backup MAD is implemented for two popular file systems, NTFS and ext2. Its snapshot requires very low storage space, 0.05% in the worst case compared to file backup solution. Also, its restoration is extremely fast that 8.9 GB of lost files can be recovered within 0.2 seconds. Even if a part of the formatted partition is overwritten, Backup MAD is still able to restore, approximately 50% more files than other scan-based recovery tools. Especially, Backup MAD for ext2 is capable of reactively reflecting up-to-date metadata at the cost of reasonable performance degradation.

In the rest of this paper, we analyze various types of disk pointers (Section 2), discuss several designs (Section 3) and implementations (Section 4) of Backup MAD, and evaluate the effectiveness of our solution with diverse experiments and metrics (Section 5).

2. POINTER-TYPE CLASSIFICATION

A storage device, especially a disk, usually has logical block address (LBA) space which is an abstraction of a complex device structure [20]. For this reason, every file system has its own *on-disk format* to map its data to the linear address space. *Partition formatting* is a way to initialize a partition state by writing an on-disk format of file system into the partition. We call it *high-level formatting* to format a partition by clearing out metadata regions only and *low-level formatting* to fill an entire region with some values. Tools like *mke2fs* or *quick-format* belong to the former and *dd* or *non-quick-format*, to the latter.

Even though various block dependencies are utilized by different file systems, they are generally classified into the following four types according to our analysis:

- *Decoding pointer* contains an offset for file system to decode on-disk structures.
- *Data pointer* indicates a set of block addresses where file data is located.
- *Index pointer* links multiple data structures designed and used for efficient retrieval of directories.
- *Metadata pointer* points to the metadata region assigned to a file.

This classification helps a user application understand complex on-disk structures of file system and retrieve useful information from it by *raw access*. The rest of this section investigates the on-disk formats of two popular file systems, NTFS and ext2, based on the classification.

2.1 Analysis of NTFS disk pointers

Master File Table (MFT) and *Index Record* are two important data structures in NTFS. MFT is a set of *MFT entries*, each of which includes metadata information on a file, such as file size, file name, permission and so on. An MFT entry contains several decoding pointers to parse attributes of a file. An attribute consists of *header* and *content*. $DATA is an attribute, the content of which is either file data or the location of file data. When file data is small, the size of which is not known to public but approximately less than 700 B, it is encoded into the content of $DATA, which is called *resident data*. Otherwise, it contains a *cluster run list* to track the location of *non-resident data*. A cluster is a basic I/O unit and is composed of multiple sectors, usually 8 sectors by default configuration. Each cluster is assigned a unique address, *Logical Cluster Number (LCN)*, in a partition. If we decode a cluster run list, we get an ordered list of <LCN, the number of clusters> pairs that track all LCNs assigned to a file. This is data pointer of a file.

Index record is a data structure to reflect directory hierarchy. It is comprised of a series of *index entries*, each of which maps a file name to a *file reference address* which corresponds to metadata pointer. Some index records are linked by index pointers to form a B-tree-like structure for the purpose of efficient lookup for a specific file name. When there is too much metadata information to be contained in an MFT entry, NTFS assigns several *non-base MFT entries* to extend the metadata region, and inserts metadata pointers pointing to the base one. Fig 1 shows an illustrative example of disk pointers used by NTFS.

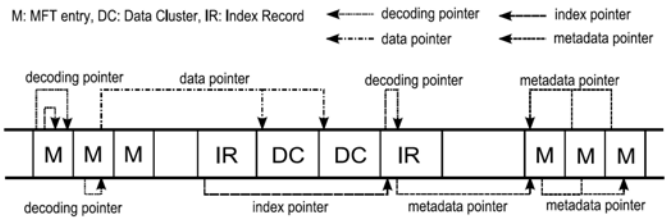

Fig 1. Disk pointer types in NTFS

2.2 Analysis of ext2 disk pointers

Inode table and *directory block* are two key data structures in ext2. The former contains a set of inodes, each of which contains metadata information on a file. An inode has an array of integers, *i_block*, to store data pointer of a file. Unlike a cluster run list in NTFS, i_block doesn't tell full block addresses of a file. Instead, ext2 uses single/double/triple indirect blocks to access a large file. They hold block addresses of either direct blocks or other indirect ones.

A *directory entry* in directory block maps a file name to an *inode number* which corresponds to metadata pointer. Each entry has *length* field for file system to determine the offset of the next entry. This field is classified as decoding pointer. Ext2 doesn't have an index pointer type since it doesn't use any special on-disk structure to efficiently retrieve directory entries; it just scans directory blocks linearly to find a file name, which accounts for low performance when there are lots of entries in a directory. Fig 2 shows an example of disk pointer types in ext2.

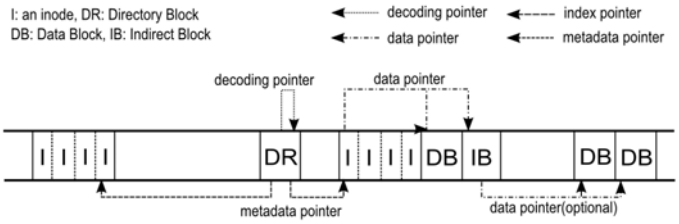

Fig 2. Disk pointer type in ext2

3. BACKUP MAD DESIGN

Fig 3 shows three components of Backup MAD with data flows. *Walker* component traverses directories specified by a user and creates a list of files to be protected from DPC. *Snapshot* component receives the list and extracts recovery information for the files by raw access. The result is saved as a file which can be submitted to other storage backend, e.g. local partitions or cloud storage such as Amazon S3 [21] and Cumulus [22]. *Restore* component either 1) copies data blocks of lost files to a spare partition having normal file system, or 2) locally updates on-disk structures by raw access to establish disk pointers between lost data blocks and file system metadata.

Backup MAD allows a user to specify which set of files to be protected or excluded.

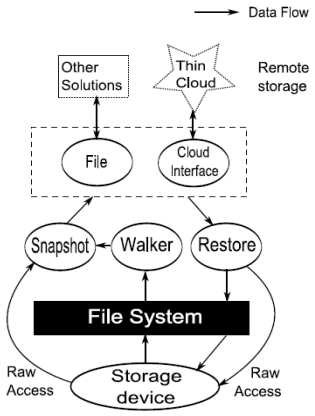

Fig 3. The architecture of Backup MAD

This feature is for satisfying various requirements of users. For example, certain users want to exclude temporary files generated during installation or cookie files saved by web browsers. Some may consider that huge software, e.g. Microsoft Office, need not be protected since it can be re-installed when some of the files are missing or corrupted. The current design doesn't contain any component to automatically select a set of candidate files, but can be improved by suggesting well-known directories such as the My Document folder to users or by dynamically profiling the access pattern of files in system to differentiate popular ones and protect them.

Backup MAD is designed to store a snapshot image either 1) as a file in a specific partition, or 2) as a resource in Amazon S3. Using the former option, we can maintain snapshot images without network access. In the latter approach, we can improve existing online backup and recovery solutions. Although the online file backup is very powerful, we believe that not all users want the most powerful protection scheme. This is mainly because 1) file backup requires very large cloud storage space, which incurs high storage and network cost, and 2) backing up only critical disk pointers is sufficient to effectively deal with accidental format or deletion at low storage cost. Backup MAD can be easily integrated with the existing solutions without much modification.

3.1 Snapshot Technique

Snapshot component extracts data pointers for a set of files and creates a *snapshot image*. The image is composed of *snapshot entries*, each of which describes recovery information on a file. To balance strong/weak points of diverse recovery solutions, we consider four requirements that a faithful snapshot image should satisfy. Two representative snapshot examples that satisfy the requirements will be given.

3.1.1 Requirements of a Snapshot Image

R1. A snapshot image should contain metadata only, but not data: Some tools like e2image [14] extract metadata region from file system, but they may reveal confidential data since certain file system embeds small-sized file data into the region, e.g. resident

data in NTFS.

R2. The space overhead of a snapshot image should be low: Some metadata is meaningful only for file system itself, but not for a user. It could be pruned for cutting down storage cost.

R3. A snapshot image should preserve directory information enough for recovery: A scan-based recovery tool or an integrity checker often fail to recover directory information.

R4. A snapshot image should provide a recovery tool with sector-level information of files: It helps an application retrieve file data by raw access.

Table 1 describes which set of requirements is satisfied by each solution. Data backup solution restores data by simply copying them into a partition, but it requires very large storage space and an additional encryption technique to keep confidentiality of data. Both of a scan-based tool and an integrity checker don't take advantage of a snapshot image but depend only on a partition state. However, the previous study [18] points out that the restoration by an integrity checker is sometimes unsafe that directory hierarchy becomes incorrect. They don't reveal any recovery information to an external tool or a user. Metadata replication technique also hides internal data, therefore, making it hard to rescue data blocks when OS or file system is totally out-of-order. Ntfs File Sector Information (NFI) is a tool used to dump information about an NTFS volume and determine which volume and file contains a particular sector [23]. It doesn't perform any recovery procedure or display the locations of resident data. Backup MAD is the only solution to meet all of the requirements.

Table 1. Requirement satisfiability. Each requirement is fully(O), partially(△), or never(X) satisfied by each approach. Some aren't applicable (-) to the solution.

Recovery Method	R1	R2	R3	R4
Data Backup	X	X	O	-
Integrity Checker	O	O	△	X
Metadata Replication	O	△	O	X
Scan Recovery	O	O	△	X
Dumping Metadata-region	△	△	△	△
Ntfs File-sector Information	-	-	-	△
Backup MAD	O	O	O	O

3.1.2 Design of a Snapshot Image

A snapshot entry should include the followings and can optionally cover additional recovery information, e.g. permission, according to a service level a user wants:

(file path, file size, data pointer)

This snapshot entry actually contains no file data (R1), minimizing leakage of private information. It also excludes most of metadata information not essential for recovery (R2). A file path inherently reflects directory hierarchy (R3) and data pointer encodes block addresses where user data is located (R4).

It is noticeable that only data pointer type is recorded in a snapshot entry. It is

mainly because we believe that decoding/metadata/index pointers are only meaningful in that they enable file system to find out data pointers of files. Hence, Backup MAD doesn't put much importance on the three types. They will be re-created by file system APIs during our restoration procedure, which is described in the following subsection.

An exemplar snapshot image in NTFS is given in Table 2. R/N indicates a type of file data: if it is 1, its type is non-resident. Data pointer is represented by either (LBA, the offset within the sector) or a list of (the number of sectors, the first LBA of the group of sectors) for resident and non-resident data, respectively. For the case of resident data, its size is so small that it is included in $DATA attribute of the file's MFT entry. In the example, 'filePath1' is resident type and its MFT entry spreads over two sectors, 6301199 and 6301200. The file data starts at 288-th byte within the MFT entry. The file data of 'filePath3' occupies 8 sectors, i.e. a cluster, starting from 47949591st sector, which is represented by [(8, 47949591)]. Runvalue is an encoded value of data pointer. We include the value in a snapshot image for the purpose of eliminating redundant calculation since it doesn't change until a partition is resized. Table 3 illustrates a snapshot image in ext2. Data pointer is composed of a list of direct blocks and indirect blocks, each of which corresponds to i_block fields in an inode. The "isDir" field indicates whether the current file is a directory or not. If it is 1, it is a directory and will be recovered by using file system API, which will be discussed in the following subsection.

Table 2. A snapshot image in NTFS

FilePath	FileSize	R/N	Data Pointer	RunValue	MFT #	MFT LBA
filePath1	496	0	(6301199, 288)	0x0	4840	6301199
filePath2	232	0	(6301201, 296)	0x0	4841	6301201
filePath3	942	1	[(8, 47949591)]	0x5b74db0131	4842	6301203

Table 3. A snapshot image in ext2

FilePath	FileSize	isDir	Inode #	Data Pointer ([Direct Blks],IndBlk,2IndBlk,3IndBlk)
filePath1	4096	1	97845	[3296952], 0, 0, 0
filePath2	9024	0	61326	[2065616 2065624 2065632], 0, 0, 0
filePath3	132777	0	91642	[3108696 3108704 3108712 3108720 3108728 3108736 3108744 3108752 3108760 3108768 3108776 3108784], 3108792, 0, 0

3.2 Restoration Technique

We propose three safety conditions which any faithful restoration technique must obey.

C1. Recovery process must not corrupt live data within file system (OS-awareness).

C2. Recovery process must not corrupt not-yet-recovered data (Ordered recovery).

C3. Repeated recovery should produce the same partition state as if there had never been a crash (Idempotence).

Let us define a few terms for further discussion. *Corrupted partition* is a partition in which DPC has occurred. *Spare partition* is in a clean state with a normal file system. A destination for to-be-restored files can be set to either a spare partition or a corrupted one.

The former case is called *Out-of-Partition (OP) restoration* and the latter *In-Partition (IP) restoration*. Each version of Backup MAD is named as *MAD-fsType*. We have designed and implemented MAD-NTFS and MAD-ext2.

Existing solutions break one or more of the safety conditions. For instance, Norton Ghost [24] restores an original partition state by overwriting a corrupted partition entirely, violating C1. For the same reason, dumping metadata (e.g. e2image) in advance and copying it back into its original place would corrupt live data. In case of a scan-based recovery tool, a spare partition other than a corrupted one is essential for recovery. If a corrupted partition is used for containing recovered data, the tool may overwrite not-yet-recovered data since it uses only file system API when creating new files, which breaks C2.

On the other hand, OP restoration satisfies all of our safety conditions. The technique reads data blocks from a corrupted partition using data pointers in a snapshot image and writes the corresponding data blocks to a spare partition with file system API. It issues only read requests to the corrupted partition, and therefore doesn't change the state of it (C3). Besides, as the technique makes use of file system API to write files, it never harms current live files (C1). Trivially, not-yet-recovered data blocks reside only in the corrupted partition, it is impossible for them to be overwritten by newly-recovered data (C2).

IP restoration directly updates disk pointers inside a corrupted partition by raw access, but not on a corrupted block in place. Consequently, a recovered file may have its corresponding index/metadata pointers in different locations (non-corrupted blocks in the corrupted partition) compared to the original one. IP restoration regards that it successfully recovers a file as long as its data pointer remains the same as before.

Algorithm 1 shows balanced usage between raw access and file system API, a key technique to IP restoration. The motivation comes from the facts that 1) the semantic behavior of file system can't be inferred from its on-disk format, and 2) on-disk format of file system may not be fully known to public. For example, NTFS doesn't reveal where to assign a new MFT entry for a new file. Any heedless update like picking up a random location would cause serious side effects. Besides, restoring all the types of disk pointers is extremely hard because the complete on-disk format of NTFS isn't known yet. To overcome the problem, IP restoration benefits from file system API to establish decoding/index/metadata pointers for lost files, whereas it relies on raw access to embed data pointers for them. As a result, it can safely modify on-disk structures in spite of the lack of the semantics and the complete on-disk format of file system. Fig 4 demonstrates an example of the balanced usage applied by MAD-NTFS.

In case of MAD-ext2, IP restoration is very simple since 1) there is no resident data in an inode, and 2) the Linux command "umount" forces in-memory data to be flushed into a storage device. The detailed procedure is described in Algorithm 2.

Algorithm 1. In-Partition Restoration after a partition formatting for NTFS
NR-List is a list to contain information on non-resident files.
R-List is a list to contain information on resident files.
1: Read a snapshot image and gather all the addresses of clusters in use
2: Modify *$DATA* attribute of *$Bitmap* (MFT_6) to mark those clusters as "in-use"
3: Reboot (for the purpose of caching the on-disk modification in memory)
For each snapshot entry in a snapshot image,
4: Read (file path) and append it to *NR-List* if it is of non-resident type
5: Using (data pointer), read (file data) and append it with (file path, file size) to *R-List* if it is of resident type
6: For each (file data, path, size) in *R-List*,
Mkdir/open "path", write "file data", truncate to "size" and close
7: For each (path) in *NR-List*,
Mkdir/open "path" and close
8: Reboot (for the purpose of flushing new MFT entries into a disk)
For each snapshot entry in a snapshot image, if the file is of non-resident type,
9: Read (file path, file size, data pointer)
10: Get an MFT number corresponding to "file path" by raw access
11: Read the MFT entry and transform $DATA attribute into non-resident type
12: Insert "data pointer" into $DATA attribute
13: Update the size field to "file size"

Algorithm 2. In-Partition Restoration after a partition formatting for ext2
1: Read a snapshot image and data pointers of all files
2: Gather all the addresses of data blocks based on the data pointers
3: Mask all the data blocks in the corresponding bitmap blocks
4: Mount the partition
For each snapshot entry in a snapshot image,
5: If the entry is of directory type, create the path by using mkdir()
6: Else create an empty file by using open()/close()
7: Unmount the partition to force any in-memory modification to be flushed
For each snapshot entry in a snapshot image,
If the entry is not of directory type,
9: Read the inode of the corresponding empty file into memory
10: Modify the in-memory inode to include the data pointer in the entry
11: Write the in-memory inode into its location within a partition
12: Mount the partition

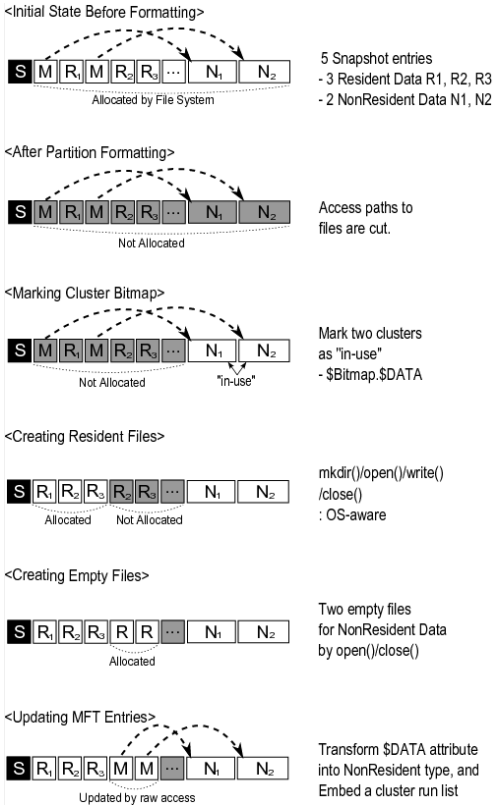

Fig 4. An example of In-Partition restoration in NTFS. S denotes a set of meta-files, M represents an MFT entry for a non-resident file, N_i is a set of data blocks belonging to a non-resident file, and R_i is an MFT entry for a resident file.

During restoration, the technique doesn't corrupt live data within file system since it makes use of file system API to allocate new MFT entries (C1). Also, it masks some bits in a bitmap structure as specified in a snapshot image to prevent any file system API such as *mkdir* and *write* from overwriting not-yet-recovered data. The order of restoration guarantees that all of lost files can be safely recovered from a corrupted partition (C2). Eventually, IP restoration transforms a lost file into a live one with few copy operations. Hence, it is very fast. Unfortunately, this technique maintains resident data only in memory during the lines 4~6 in the algorithm, so system crash would purge out data. To make it robust, it is necessary to save the volatile data into other persistent storage, which is one future direction in our work.

4. IMPLEMENTATION

In this section, we present implementation details specific to file system. We have

addressed challenges in the implementation of Backup MAD. The existence of other challenges implies a couple of hints for improving the current prototype in the next release.

4.1 MAD-NTFS

To extract a data pointer of a file, MAD-NTFS finds out where the MFT entry of the file is. MAD-NTFS utilizes only raw access to a NTFS partition since it is a user-level application incapable of invoking kernel-level API. First of all, it gets 'MFT number → LBA' mapping from \$DATA attribute in MFT_0 (also called \$MFT), where MFT_i is assumed to be *i*-th MFT entry. The attribute tracks the clusters that contain bitmap structures. Each bit of the bitmap indicates whether the corresponding LCN is used for MFT. In this way, MFT0 can track all of MFT entries in a partition. When there are a lot of files in a partition, clusters used for MFT allocations can be scattered throughout the partition. For the case, the start address of each split MFT can be found by referring to MFT_0. By dividing an MFT number by cluster size, MAD-NTFS is able to locate which cluster contains the MFT entry.

The next step is to find 'file path → MFT number' mapping. This is acquired by retrieving index records from the top-most, i.e. MFT_5, and following decoding pointers, index pointers and metadata pointers iteratively. An example of the retrieval is illustrated in Fig 5. A file path is split into n-tuple of directory names (d_1, d_2, …, d_n), and a file name f_1. In the figure, the file path consists of *(Document, My Picture)* and *g.jpg*. An MFT entry corresponding to d_i usually has at least one of two attributes, \$INDEX_ROOT or \$INDEX_ALLOCATION. They contain either metadata pointers to files in d_i, or index pointers to other index records. Backup MAD compares d_{i+1} with the key from each index entry in the directory. If it finds an exact match, i.e. $key == d_{i+1}$, it can get the MFT number of d_{i+1} and search for a next match with d_{i+2} at a deeper level. Oth-

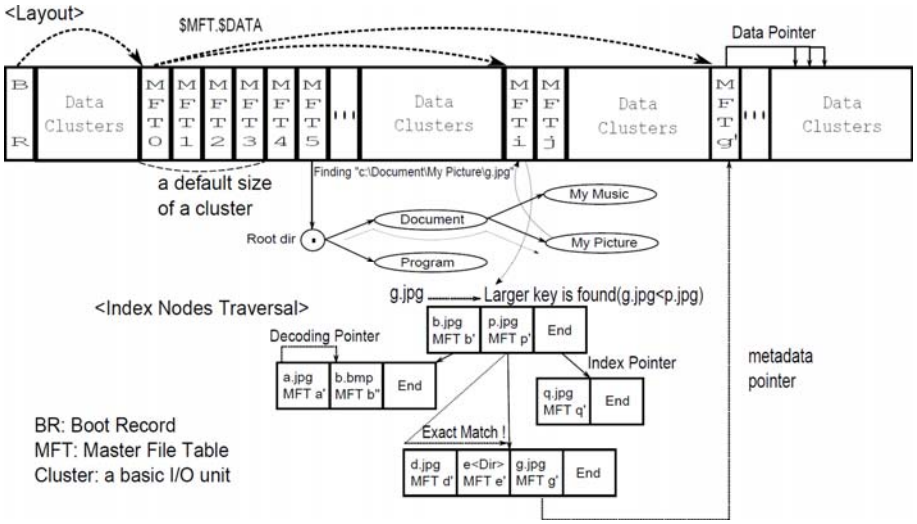

Fig 5. An example of data retrieval in NTFS

erwise, it scans index entries until it discovers a larger key, i.e. $key > d_{i+1}$, and further follows the index pointer to find an exact match. The procedure is repeated until an index entry of f_1 is found. Finally, the MFT number of f_1 is known and the location of the corresponding MFT entry can be calculated as we previously described.

Repeating the entire procedure for each file incurs much overhead because the retrieval generates a lot of disk accesses. MAD-NTFS uses *intermediate mapping cache* to eliminate redundancy in lookup operations: it is memory cache that contains information for mapping a filename to the MFT entry of the parent directory of the file. The technique maintains a key-value dictionary in memory. A key is in the form of $\{d_1, d_2, \ldots, d_k\}$ $(1 \le k \le n)$ and a value in the form of an MFT number. If any two files, X_1 and X_2, share the same directory tuple, extracting X_2's data pointer after accessing X_1 requires only two disk accesses with the help of the cache.

The prototype of MAD-NTFS is purely written within 2,000 lines of python script. If we exclude some part specific to NTFS, the rest can be reused for other versions since the interpretation language is inherently very portable. *Win32file* module in python supports a subset of *win32API* such as CreateFile, SetFilePointer, ReadFile and WriteFile, which are essential for accessing NTFS partition in a raw mode.

4.2 MAD-ext2

To extract recovery information on a file, MAD-ext2 needs two mapping information, 'file path → inode number' and 'inode number → LBA'. *ext2fs* library [14] wraps various utility functions to access a raw partition initialized by ext2 file system. It is very trivial to extract the mappings with the library. Simply, MAD-ext2 gets an inode number of a file by invoking the *stat* system call. It then passes the number to the library functions to modify the inode corresponding to the number. All codes parts are written in python and C. Ext2-specific part is written in C and the rest in python, mostly borrowed from MAD-NTFS.

Linux has event notification mechanism, called *inotify*, that delivers events related with file operations to an application. This feature enables an application to watch any modification to files or directories without help of any specialized device driver. The initial implementation of MAD-ext2 was very similar to MAD-NTFS which periodically creates backups of a snapshot image. Although this approach has very low failure-free overhead, a snapshot image may be in an inconsistent state between backup periods.

The improved MAD-ext2 takes advantage of inotify mechanism to reflect modified data pointers in a snapshot image as soon as possible. It receives events from file system and enqueues them in an event queue. If the event type is *create* or *modify*, MAD-ext2 updates the status of the corresponding file as *dirty*. Otherwise if the event type is *delete*, the corresponding snapshot entry residing in memory is invalidated. MAD-ext2 spawns a separate thread that behaves much like *pdflush* daemon: it periodically wakes up and extracting data pointers of the files in dirty file list.

Still, this design can't completely prevent a snapshot image from being in an inconsistent state because the processing of extracting up-to-date disk pointers can be delayed until the separate thread wakes up. The design rationale is for buffering frequent updates to the same file within short time interval. Otherwise, MAD-ext2 would have extracted the data pointer of a file multiple times even though all of them but the last one will be

out-of-date, hence useless. This buffering technique greatly reduces the overhead of taking a snapshot image even though a foreground task performs heavy I/Os or computations.

4.3 Implementation Challenges

The current prototype of Backup MAD lacks some functionalities because 1) it is incapable of invoking device-driver level API, and 2) certain corner cases are ignored for simplifying our restoration model. We describe a few difficulties in completing Backup MAD and suggest possible workarounds or solutions in the following.

4.3.1 Synchronization Problem

It is observed that some modifications by IP restoration are invalidated by corresponding dirty pages cached in memory and later flushed into a storage device. Backup MAD learns from the observation and makes a detour to deal with the synchronization problem.

In case of MAD-ext2, it can unmount or mount a specified partition with root privilege. The feature makes it easy to forcibly flush dirty pages and reload any modified on-disk value in memory. On the other hand, MAD-NTFS can't remount a partition directly since only device driver can perform the job in Windows environment. Instead, MAD-NTFS produces the same effect with *shutdown* command. When Windows reboots, it initially caches a small amount of critical metadata modified by MAD-NTFS from an NTFS partition.

However, the latter technique is heavily dependent on the fact that NTFS doesn't flush clean pages when it shutdowns. Certain file system like ZFS flushes buffer-cached pages regardless of their dirtiness [25], which would invalidate any on-disk modification by raw access. It is a major drawback of our approach, which must be solved to apply IP restoration technique generally to other types of file system.

4.3.2 Out-of-bounds Restoration

Unlike OP restoration, IP restoration transforms the "old" partition into the "new" partition. In this case, if a formatted partition becomes smaller than its original size, data pointers of lost files may point to blocks beyond the old partition. If the case isn't considered, access paths to the restored files will be cut by the sanity checking of file system. Copying those blocks into the new partition would be a solution, but it might violate two safety conditions C2 and C3 without careful update. To resolve the problem, it is necessary to copy those blocks to only free blocks both in the "old" and "new" partition and adjust the data pointers to point to them. It is one of our future research directions.

4.3.3 Limitations of inotify mechanism

Even though the inotify mechanism delivers in-kernel events to a user application, there are two potential problems in the design of MAD-ext2. One is security vulnerability.

If a malicious user changes metadata of a file very frequently by creating/deleting it within a short interval, the data pointer would become so stale that it may happen to point to data blocks belonging to other files. At this point, if the user requested its deleted file to be recovered, Backup MAD could provide the content of other files to the user. To fix this vulnerability, when requested a restoration of a file, Backup MAD should use barrier to ensure that the target file has an up-to-date data pointer, which isn't implemented in Backup MAD yet.

The other problem is performance degradation. MAD-ext2 must retrieve up-to-date data pointers on disk by raw access even though they probably exist in memory. It has no interface to get the data in kernel memory. Also, the overhead of frequent context switches accounts for the high execution time as shown our evaluation section 5.5.

In fact, the two weaknesses are inherent for a user-level application. If high performance or high security criteria are required, a kernel-level solution will be better since it can directly access up-to-date metadata in memory and minimize context switching to handle file operation events.

5. EVALUATION

We have used two types of datasets. The one is synthetic dataset gathered from a PC used for two years. They are classified into three groups as described in Table 4, to compare performance metrics between MAD-NTFS and other solutions. The other type is representative dataset generated by various well-known benchmarks such as postmark, bonnie++, sysbench and kernel-compile. They are used for testing MAD-ext2 that reactively updates a snapshot image. A machine used for experiments has 3 GHz CPU, 2 GB memory and two disks. The one disk maintains full copies of original datasets for the purpose of checking byte-level equivalence between an original file and a recovered one. The other is for the evaluation of each recovery solution.

Table 4. Characteristics of synthetic datasets

	Dataset1	Dataset2	Dataset3
# of dirs	9,323	7	1
# of files	106,072	566	11
Median	4 KB	18 MB	663 MB
Total size	15.4 GB	17.3 GB	8.9 GB

5.1 Field-Aware Fault Injection

In this evaluation, we confirm that even a small corruption can result in large data loss and that metadata replication scheme and integrity checker sometimes fail to protect critical metadata. For concrete analysis, we chose NTFS and CHKDSK as representative solutions for metadata replication and integrity checker, respectively. We have used our fault injection framework, Field-Aware Fault Injection (FAFI), to test the two schemes. It generates many copies of a specified file and injects a fault to each of them for every combination of an attribute and an injection position. From what we understand, previous fault injection frameworks like Type-Aware Pointer Corruption [1] mainly attack index

pointers and metadata pointers. However, when retrieving file data, file system observes decoding pointers more frequently than other types. To complement the previous studies [1, 2, 3, 11, 19], we focus on the cases where various decoding pointers are corrupted.

Table 5 shows diverse types of data loss when each fault is injected to an attribute (as explained in Table 7) of a regular file. NTFS and CHKDSK fail to restore regular file data in 13 cases out of 17 ones. For instance, a bit flipping in a residency bit truncates the whole file data. Corrupting a standard information attribute, $STD_INFO, also makes the file invisible from a directory.

In the second experiment, faults are injected to the MFT headers (as explained in Table 8) of important metafiles reserved by NTFS. They usually affect the entire partition. Table 6 shows that NTFS successfully recovers metafiles in 36 scenarios (√, ⊙) by its internal metadata replication. Although NTFS fails to recover in 11 cases (⊖,⍰,⊕), CHKDSK can still repair the corrupted metafile. It seems that metafiles are carefully managed by NTFS and CHKDSK due to their importance, which is also stated in [26]. However, one fault injection scenario (X) bypasses the two protection schemes and harms the entire partition. This is another example of *ineffective replica management* [18].

From our observations, it is confirmed that a trivial corruption in decoding pointer propagates the fault in a form of data loss or partition loss. Neither metadata replication nor integrity checker can't perfectly prevent data loss caused by pointer corruptions. On the other hand, Backup MAD is able to successfully recover in all the fault injection scenarios. The difference originates from the fact that Backup MAD relies on high-level recovery information enough for correct restoration. Both of NTFS and CHKDSK also take advantage of replicated metadata information, but it seems that the replica may give false or incomplete information to them.

The two experiments don't show a full list of possible scenarios, though. We anticipate that FAFI can provide complete analysis of failure behaviors of NTFS and CHKDSK, which is a part of our on-going work.

Table 5. Status of a file after fault injection. Fault injection to the attributes of a regular file, (a) residency bit, (b) attribute length, (c) content size, (d) content offset, (e) start VCN, (f) end VCN, and (g) compression unit size. Each symbol represents the status of a file after corruption, -: not applicable, X: not existent, ⊗: existent but not accessible, ◎: accessible but no data, and √: correct data.

Attribute	(a)	(b)	(c)	(d)	(e)	(f)	(g)
$STD_INFO	X	X	Ⓧ	√	-	-	-
$FILE_NAME	Ⓧ	Ⓧ	√	√	-	-	-
$DATA(R)	Ⓞ	Ⓞ	Ⓞ	Ⓞ	-	-	-
$DATA(N)	Ⓞ	Ⓧ	-	-	Ⓧ	Ⓧ	√

Table 6. Status of metafiles after fault injection. Fault injection to the MFT header of metafiles, (a) signature, (b) fixup value, (c) LSN, (d) sequence value, (e) hard link count, (f) the offset to the first attribute, (g) flags, and (h) the size of an MFT entry. Each symbol represents the status of a partition after corruption, √: need not CHKDSK, ⊖: accessible after CHKDSK, ?: labeled after CHKDSK, ⊕: labeled and accessible after CHKDSK, ⊙: labeled and accessible but need CHKDSK, and X: not recoverable by CHKDSK.

Metafile	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
$MFT	√	√	√	√	⊙	√	X	√
$Volume	√	√	⊙	√	⊙	√	⊙	√
$AttrDef	√	⊙	√	⊙	√	⊙	√	⊙
. (Root Dir)	⊖	⊖	√	⊖	√	⊖	?	⊕
$Bitmap	⊕	⊕	√	⊕	√	⊕	√	⊕
$Boot	⊙	⊙	√	⊙	√	⊙	√	⊙

Table 7. Fault-injected position information for Table 5. "Offset" indicates the byte address starting within the attribute header.

Attribute header	Type	Offset	Description
Residency	Both	8	The type of this content. If it is 1, the type of content is non-resident.
Attribute length	Both	4~7	The length of (header + content)
Content size	Resident	16~19	The length of this content
Content offset	Resident	20~21	The relative offset to the start of the content within this attribute
Start VCN	Non-Res	16~23	The start of file offset corresponding to the run list of this content
End VCN	Non-Res	24~31	The end of file offset corresponding to the run list of this content.
Compression unit	Non-Res	34~35	The unit of compression of this content. If it is 4, 2^4 clusters is the basic unit.

Table 8. Fault-injected position information for Table 6. "Offset" indicates the byte address starting within the MFT header.

MFT header	Offset	Description
Signature	0~3	If it is 'F', 'I', 'L', 'E', this MFT entry is valid.
Offset to fixup array	4~5	The relative offset to the start of fixup array used for ensuring integrity of a sector within this MFT
LSN	8~15	The relative offset within $LogFile for the purpose of logging transactions
Sequence Value	16~17	Used for calculating file reference address
Hard link count	18~19	The number of hardlinks connected to this MFT entry
Offset to the first attr	20~21	The relative offset to the first attribute within this MFT entry

5.2 Recoverability

Backup MAD saves only a small part of metadata. As a result, it can't deal with user data corruption or disk death problem. We clarify the capability of Backup MAD to clarify its capability to deal with data loss. Backup MAD can recover lost files from the following cases:

· *Data blocks of a deleted file are not overwritten yet*: Most file systems delete a file by modifying disk pointers only, leaving data blocks alive.

· *Metadata of a file is corrupted*: This type of corruption is simulated by our fault injection framework, FAFI. Backup MAD is able to restore a lost file from all of the fault injection scenarios in the previous evaluation.

· *Partition table is corrupted*: Corrupting a partition table often makes the partition unmountable. Still, all data blocks are accessible and recoverable by Backup MAD.

· *High-level partition formatting is performed*: It is the most extensive pointer corruptions since it cuts access paths to all of files in a partition. In ext2, mke2fs clears out all inode tables but all of lost files are recoverable since there is no resident file. In NTFS, although some user data resides in metadata region, all of the data is recoverable since quick-format clears out only a few metafiles.

However, Backup MAD can't recover lost files from the following cases:

· *Data blocks of a deleted file are overwritten*: Backup MAD doesn't protect user data corruption. Secure deletion [27] belongs to this case.

· *A disk drive is physically damaged*: Backup MAD and any scan-based recovery tool can't handle this type of data loss. File backup and block-level replication (e.g. RAID) are the only solutions to this case.

· *Low-level partition formatting is performed*: dd utility or non-quick-format overwrite every block in a partition with some value, which makes it impossible to restore lost files by Backup MAD.

5.2 Restoration Overhead

MAD-NTFS generates a compact snapshot for each dataset, 14 MB, 59 KB and 2 KB, respectively, as shown in Fig 6. Simply coalescing MFT entries into a binary file produces an output at least 86% larger than the corresponding MAD-NTFS snapshot, and has a potential danger of revealing resident data in the image, which doesn't satisfy R1.

Restoration time is also an important metric since fast recovery would diminish *Mean Time To Repair* when data loss occurs. We test five recovery solutions, 1) *Online* (internet) recovery solution, 2) *Local* solution which copies original files from a spare partition, 3) *Scan-based* recovery solution, 4) *OP* restoration, and 5) *IP* restoration.

Table 9 shows that IP restoration is the fastest solution, especially when the number of files in a dataset is very small. Even in the worst case where there are a lot of small files, it can restore whole files about 2 times faster than local copying because its recovery time is dependent only on the number of files in a dataset.

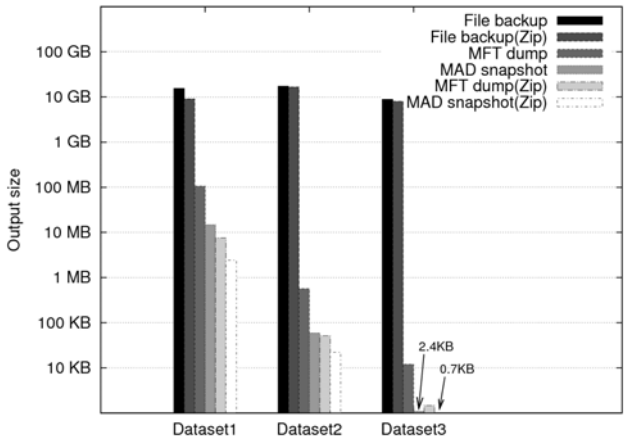

Fig 6. Storage overhead. (Zip) means that the output is compressed by bz2.

Table 9. Restoration time (in seconds)

	Dataset1	Dataset2	Dataset3
Online	100,620	135,424	75,704
Local	1,080	480	142
Scan-based	89,940	89,940	89,940
MAD(OP)	2,076	781	369
MAD(IP)	534	2.36	0.16

5.3 Recovery Success Rate

Let us define *Recovery Success Rate (RSR)* to be the rate of files that successfully recover and pass our equivalence check. For normal configurations, OP restoration satisfies all of safety conditions since it restores files on a spare partition other than a corrupted partition where lost data resided. However, if the destination of the result of OP restoration is intentionally set to a corrupted partition, it would violate our safety conditions, which eventually lessens RSR. Fig 7 shows the result of changing the location of restoring recovered files to the corrupted partition and performing OP restoration technique. The RSR of OP restoration is low for two main reasons: 1) a lot of resident files were overwritten by newly-allocated MFT entries due to *open* system calls, and 2) some non-resident files were overwritten by newly-allocated index records due to *mkdir* system calls, proved by the magic number of the metadata structure. On the other hand, IP restoration is able to restore all of lost files from and into a formatted partition, which isn't achievable by OP restoration or scan-based recovery tool. The graph shows the effectiveness of a balanced usage between raw access and file system API.

Fig 8 shows how robust Backup MAD is against extensive corruptions. To check how many files can be restored from a partially-corrupted partition, the partition is incrementally overwritten on purpose; RSR is calculated in each experiment. When 12% of the partition is corrupted, scan-based recovery tool loses half of original files, which

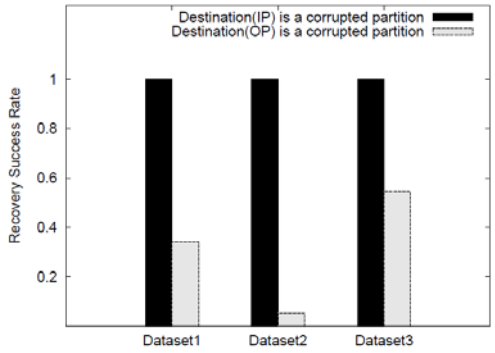

Fig 7. Effectiveness of IP restoration

means a significant fault propagation. We presume that the extensive corruptions happen to destroy critical disk pointers and make it impossible for the tool to infer block dependencies correctly. On the other hand, Backup MAD is still able to restore 95% of lost files because of using correct information in a snapshot image. Backup MAD guarantees graceful degradation [28] of RSR since it minimizes fault propagation.

When compared to scan-based recovery tool, Backup MAD outperforms it in terms of recovery success rate and restoration time (mentioned as the result of evaluation in the previous subsection). Backup MAD is able to restore more files correctly than scan-based recovery tool in a much shorter time. This gap is mainly due to the tool's lack of a snapshot information. It is expected that the scan-based recovery tool could provide more accurate restoration if its procedure were based on a snapshot image.

Even though Backup MAD guarantees correct restoration, it can't recover files, the data blocks of which are corrupted. In our experimental configuration, as corruption factor becomes larger than 0.45, the RSR of Backup MAD sharply decreases; a lot of small files in a dataset had been overwritten by intentionally copying new files to the corrupted partition. The threshold value, i.e. 0.45, may vary depending on the partition state and corruption scenario but is usually higher than that of scan-based recovery tool.

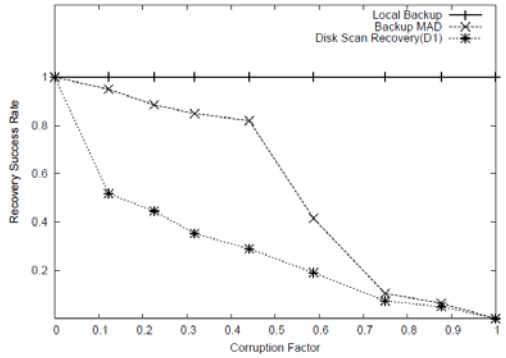

Fig 8. Comparison in RSR degradation. Corruption factor is a ratio of corrupted size to partition size

5.4 Caching intermediate mapping

Fig 9 shows that caching intermediate mappings, as explained in Section 4.1, effectively saves time to take a snapshot image. The y-axis means the time taken to extract data pointers of all the files in a dataset, normalized to the case where the snapshot component utilizes the intermediate mapping cache. As shown in Table 4, dataset1 has a lot of directories and files that share common parents in most cases. Consequently, it benefits from the intermediate mapping cache by eliminating redundant lookups. In this case, more than 90% of entries are retrieved from the mapping cache, which contributes to the reduction in time to take a snapshot image. In the case of using datasets 2 and 3, there isn't much performance benefit since the average number of files in a directory is small.

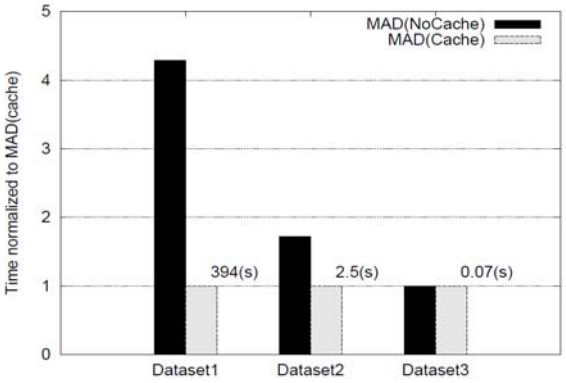

Fig 9. Benefit of intermediate mapping caching

5.5 Reactive Snapshot

As a representative benchmark, Postmark [29] is used for testing MAD-ext2. The benchmark simulates the I/O workload usually observed by ISP mail servers: it frequently creates/updates/deletes a lot of small files, therefore being regarded as the most adverse opponent to MAD-ext2 that utilizes the inotify mechanism to reflect the changes in metadata of a file as soon as possible.

To conduct a performance study for MAD-ext2, Postmark is executed under different environments. At first, we run Postmark without additional mechanism, which is the baseline for comparison. Then, by turning on the inotify mechanism only, the overhead of capturing and delivering file operation events to an application will be measured. Finally, MAD-ext2 fully operates by recording those inotify events and reflecting them in a snapshot image. Two period values, 5 and 0.1 (in seconds), are used for experiments. The settings of other configuration parameters are described in Table 10.

As shown in Fig 11, the overhead caused by MAD-ext2 is mainly due to the inotify mechanism. When period value is set to 5 seconds, extracting data pointers and updating a snapshot image makes execution time only 5~7% longer than that of the "inotify only" mechanism. But, as the period value becomes lower, updating a snapshot image accounts for the large fraction of execution time. It is because 1) a separate thread of MAD-ext2 consumes many CPU resources to periodically scan the dirty list and manage an event queue, and 2) frequent updates to the same file make independent disk I/O requests to extract up-to-date data pointers. However, when there are a lot of files in a dataset,

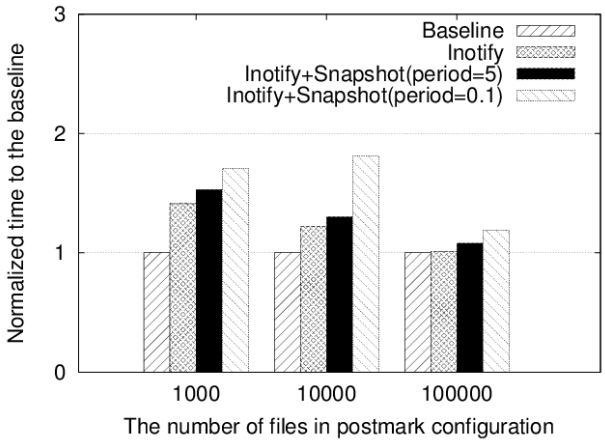

Fig 11. Analysis of MAD-ext2 overhead for Postmark

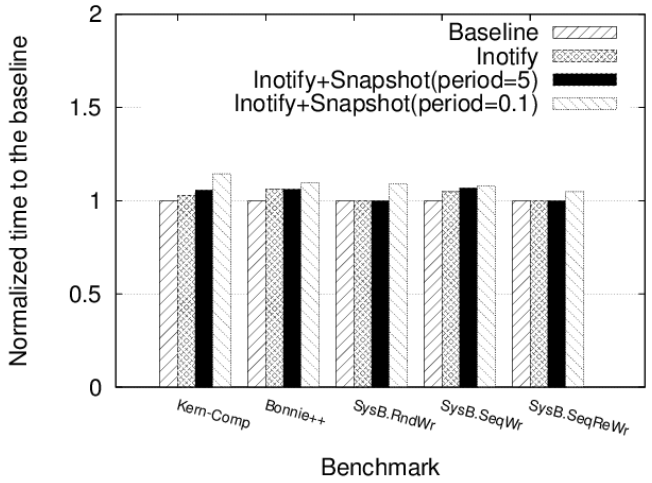

Fig 10. Analysis of MAD-ext2 overhead under various workloads

MAD-ext2 relatively incurs less overhead even if the period is set low, since the rate of in-memory modification of files becomes lowered due to the large working set generating disk I/O requests.

Unlike postmark, other benchmarks [30, 31] don't suffer from the short period value since they infrequently or never create/delete files within the period. Fig 10 shows that MAD-ext2 adds only 4~11% to the execution time of the baseline when the period value is set to 0.1 seconds.

To take a deep look at the dependency between accuracy and the period value, we plot a time-series of the number of snapshot entries during Postmark run in Fig 12. As Postmark frequently modifies file system state within short time interval, the period value directly affects the *staleness* of a snapshot image. We define the staleness as the sum of the number of snapshot entries pointing to deleted files and the number of live files not included in a snapshot image. After an experiment, the staleness of a snapshot image eventually reaches 0. But, in the middle of an experiment, a snapshot image can sometimes have a high staleness according to the period value. For example, when the period value is 1, the staleness of a snapshot image at 2.14 seconds is 35.

We can learn from the time-series graphs and performance graphs that choosing the period value implies a tradeoff between accuracy and performance. When the period is short, the staleness of a snapshot image would be low but at the cost of performance de-

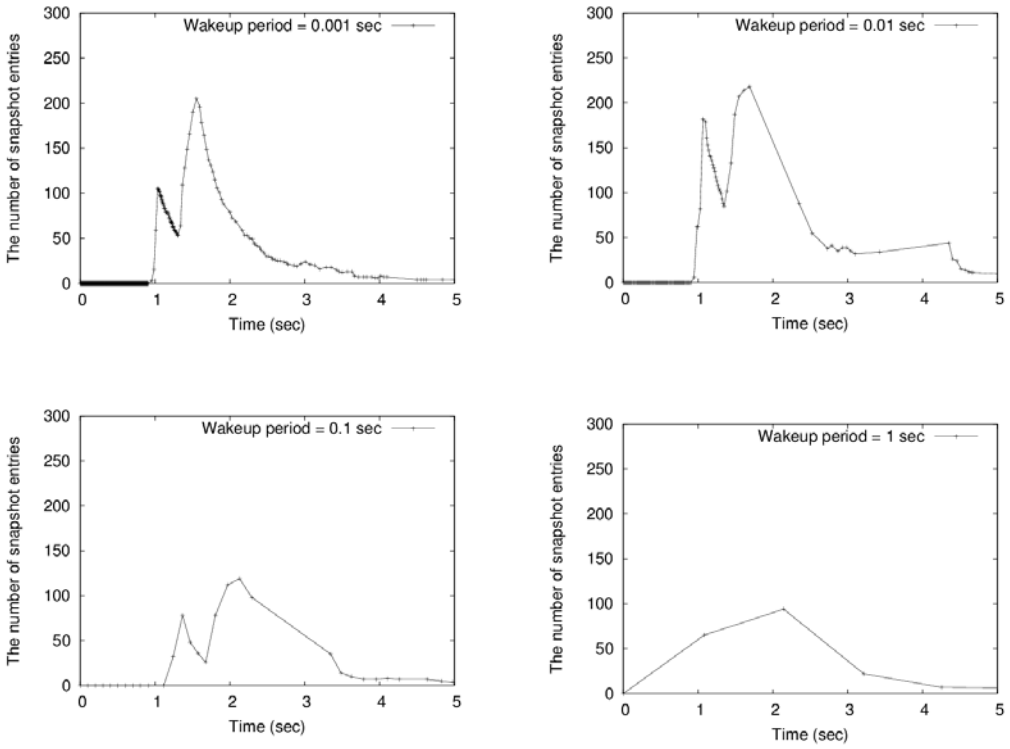

Fig 12. Time series of the number of snapshot entries during Postmark run

gradation of foreground tasks. On the contrary, the high period value limits the consumption of CPU and I/O resources, guaranteeing better performance of foreground tasks, but making it possible to have a stale snapshot image at some points.

Hence, balancing the two metrics, accuracy and performance, is very important for effectively implementing a reactive scheme to take a snapshot image at user level. One of our remaining research directions is to find the most appropriate period value in a systematic way.

Table 10. Benchmark configuration

Workload	Configuration
Postmark	# of files ∈ {1,000, 10,000, 100,000}, # of transactions = 100,000, filesize ∈ [500 B, 9.7 KB], (# of reads)/(# of appends)=5, (# of creates)/(# of deletes)=5
Kernel-compile	make –j 2 bzImage (the rest with default compile options)
Bonnie++	Performing Create/Write/Rewrite/Read for a 4 G file
Sysbench	File-test-mode ∈ {rndwrite, seqwrite, seqrewrite}, # of files = 128, filesize = 2 MB, numThreads=4

6. RELATED WORK

Snapshot™ [32] maintains point-in-time images of Write Anywhere File Layout (WAFL) file system. When data loss occurs, an administrator can revert the file system to its old image. However, the technique heavily relies on WAFL's specific features. WAFL file system inherently preserves data blocks not to be overwritten by others since it resembles log-structured file system. It is very easy for WAFL to checkpoint its pointers and to switch into a specific set of disk pointers, which isn't implemented in most commodity file systems. On the other hand, Backup MAD independently operates at user level above file system and enables commodity file systems to modify a set of disk pointers into another. From a different standpoint, we can say that Backup MAD permits using snapshot technology to file system that doesn't support it. Even if critical metadata of file system is totally corrupted, Backup MAD can still rescue live data blocks from the partition or revert to an old image as long as data blocks are preserved.

Debugfs is a file system debugger for ext2, which makes it possible to undelete a file interactively [14]. If a user selects an inode number among what the debugger suggests, it recovers the specified data with a new file name. But the tool isn't suitable for restoring a lot of files simultaneously since it violates one of our safety conditions, C2. Moreover, the debugger can't recover lost files from a formatted partition because mke2fs clears out all the inode tables that should be retrieved for recovery. In contrast, Backup MAD restores lost files simultaneously in a safe way even after partition formatting is performed.

LifeBoat [33] designs *local restoration* algorithm to solve the problem of having a backup on the same drive the solution is trying to restore to. Its goal is equivalent to that of IP restoration. The major difference is that local restoration relies on the conversion process between NTFS and FAT32; the technique converts a corrupted partition from

NTFS to FAT32, restores lost files by using raw access and re-converts the partition from FAT32 to NTFS. The paper explains that the conversion was necessary due to the incomplete write-support for NTFS driver in linux. However, Backup MAD doesn't entirely rely upon raw access to modify the corrupted partition. Due to balanced usage between file system API and raw access, IP restoration is able to restore lost files, not requiring the partition to be converted to other well-known types, minimizing the overhead of data relocation. As a result, IP restoration doesn't necessarily recover complex on-disk structures such as decoding pointers or index pointers. It only focuses on correct restorations of data pointers since other types of pointers will be re-created by file system API. Besides, while LifeBoat needs modifications to device driver, Backup MAD is a user-level application which enhances deployability.

Crystal [34] is a corruption repair framework to improve a file system checker by redefining the problem of integrity checking as a global optimization problem. The concept of *structure* is equivalent to disk pointer and *structural corruption*, to DPC. The solution suggests to leverage a file system snapshot, which is very similar to the basic idea of Backup MAD. However, the paper doesn't discuss how to design a snapshot image and a concrete recovery algorithm based on it. On the contrary, Backup MAD clearly specifies the way to organize a snapshot image and to restore lost data based on it.

7. CONCLUSION AND FUTURE WORK

Backup MAD provides a new concept of backup to save disk pointers and restore them quickly, accurately, and cost-effectively. It provides DPC-tolerance for commodity file systems. As a result, *only corrupted blocks will be dead, never affecting others*. Of course, our solution doesn't protect user data from being corrupted, but it is expected that it can be easily integrated with existing backup solutions with little space overhead.

One direction in our future work will be to apply our Backup MAD approach to DBMS. It is known recently that some DBMS can't deal with pointer corruptions effectively [8]. They may lose large tables even for a small number of bit corruptions. We anticipate that Backup MAD can ensure DPC-tolerance to any DBMS, on-disk format of which is known to the public.

REFERENCES

1. L. N. Bairavasundaram, M. Rungta, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Analyzing the Effects of Disk-Pointer Corruption," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN'08)*, Anchorage, Alaska, June 2008.
2. S. Subramanian, Y. Zhang, R. Vaidyanathan, H. S. Gunawi, A. C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau and Jeffrey F. Naughton, "Impact of Disk Corruption on Open-Source DBMS," In *Proceedings of the 26th International Conference on Data Engineering (ICDE'10)*, Long Beach, California, March 2010.
3. A. C. Arpaci-Dusseau, "Model-based failure analysis of journaling file systems," In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 802 – 811, Washington, DC, USA, 2005. IEEE Computer

Society.

4. J. Elerath. “Hard disk drives: the good, the bad and the ugly!,” *ACM Queue.*, 5(6):28–37, September 2007.
5. L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “An analysis of data corruption in the storage stack,” In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST’08)*, pages 1 – 16, Berkeley, CA, USA, 2008. USENIX Association.
6. H. S. Gunawi, C. Rubio-Gonzalez, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, “Eio: Error handling is occasionally correct,” In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST’08)*, San Jose, California, February 2008.
7. W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, “Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure characteristics,” In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST’08)*, San Jose, California, February 2008.
8. B. Schroeder and G. A. Gibson, “Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?,” In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST’07)*, San Jose, California, February 2007.
9. Ontrack Data Recovery, “http://www.ontrackdatarecovery.com/understanding-data-loss”, Understanding data loss.
10. Protect Data, “http://www.protect-data.com/information/statistics.html”, Statistics about leading causes of data loss.
11. H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Improving file system reliability with i/o shepherding,” In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP’07)*, pages 293–306, New York, NY, USA, 2007.
12. G. Sivathanu, C. P. Wright, and E. Zadok, “Ensuring data integrity in storage: techniques and applications,” In *Proceedings of the 2005 ACM workshop on Storage security and survivability (StorageSS’05)*, pages 26 – 36, New York, NY, USA, 2005. ACM.
13. Microsoft, “http://support.microsoft.com/kb/315265,” CHKDSK.
14. E2fsprogs, “http://e2fsprogs.sourceforge.net/,” e2image, e2fsck, ext2fs library, debugfs.
15. Active@ Data Recovery Lab, “http://www.ntfs.com/disk-scan.htm”.
16. Final Data, “http://www.finaldata.com”, Data Recovery Solutions.
17. V. Henson, Z. Brown, T. TS’o, and A. van de Ven, “Reducing fsck time for ext2 file systems,” In *Linux Symposium*, 2006.
18. H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “SQCK: A Declarative File System Checker,” In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI ’08)*, San Diego, CA, December 2008.
19. V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Iron file systems,” In *Proceedings of the twentieth ACM Symposium on Operating Systems Principles (SOSP’05)*, pages 206 – 220, New York, NY, USA, 2005. ACM.
20. C. Ruemmler and J. Wilkes, “An introduction to disk drive modeling,” *IEEE Com-*

puter, 27:17–28, 1994.
21. Amazon Web Service, "http://aws.amazon.com/s3/," Amazon S3.
22. M. Vrable, S. Savage, and G. M. Voelker, "Cumulus: Filesystem Backup to the Cloud," In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, San Francisco, CA, February 2009.
23. Microsoft, "http://support.microsoft.com/kb/253066," Ntfs File sector Information (NFI) utility.
24. Norton Ghost, "http://www.symantec.com/norton/ghost".
25. Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: a ZFS case study," In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, San Jose, CA, February 2010.
26. D. A. Solomon, "Inside Windows NT," *Microsoft Programming Series*, Microsoft Press, 2nd edition, 1998.
27. S. Bauer and N. B. Priyantha, "Secure data deletions for linux file systems," In *Proceedings of the 10th USENIX Security Symposium (Sec'01)*, Washington, D.C., August, 2001.
28. M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Improving Storage System Availability with D-GRAID. In *Proceedings of the Third USENIX Symposium on File and Storage Technologies (FAST'04)*, San Francisco, CA, March 2004.
29. J. Katcher, "PostMark: A New File System Benchmark", *Technical Report 3022, NetApp*.
30. T. Bray and R. Coker, "http://www.coker.com.au/bonnie++", *Bonnie++ Document*
31. A. Kopytov, "http://sysbench.sourceforge.net/docs", *SysBench manual*
32. NetApp, "http://media.netapp.com/documents/snapshot.pdf," Network Appliance Snapshot Technology, 2004.
33. T. Bonkenburg, D. Diklic, B. Reed, M. Smith, M. Vanover, S. Welch, and R. Williams, "Lifeboat: An autonomic backup and restore solution," in *Proceedings of the 18th Large Installation System Administration (LISA'04)*, pages 159–170, Berkeley, CA, USA, 2004. USENIX Association.
34. H. Wang, B. He, V. Prabhakaran, and L. Zhou, "Crystal: The Power of Structure Against Corruptions," In *Workshop on Hot Topics in System Dependability (HotDep'09)*, Estoril, Lisbon Portugal, June 2009.

Young Jin Yu (柳榮振) received his B.S. degree in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2006. He received his M.S. degree in 2008 and is currently a Ph.D. candidate in the School of Computer Science and Engineering at Seoul National University. His research focuses on evolving storage system, e.g. by improving reliability of file system, eliminating semantic gaps in storage stacks, or granting new capabilities to storage clients.

Dong In Shin (申東仁) received his B.S. degree in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2003. He received his M.S. degree in 2005 and is currently a Ph.D. candidate in the School of Computer Science and Engineering at Seoul National University. His research interests are in the areas of storage system, distributed computing system and I/O system analysis.

Hyeong Seog Kim (金亨錫) received his B.S. degree in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2003. He received his M.S. degree in 2005 and is currently a Ph.D. candidate in the School of Computer Science and Engineering at Seoul National University. His research interests are in the areas of Cloud and Grid computing, distributed storage, social networks, and energy efficiency.

Hyeonsang Eom (嚴炫相) received his B.S. degree in Computer Science and Statistics from Seoul National University (SNU), Seoul, Korea, in 1992. He received his M.S. and Ph.D. in Computer Science from the University of Maryland at College Park, Maryland, USA, in 1996 and 2003, respectively. He worked as an intern in the Data Engineering Group at Sun Microsystems, USA, in 1997. Before becoming a professor at SNU in 2005, he worked as a senior engineer in the Telecommunication R&D Center at Samsung Electronics. He is currently a professor in the School of Computer Science and Engineering at SNU. His research interests are in the areas of Cloud Computing, High Performance Storage Systems, Energy Efficient Systems, Fault Tolerant Systems, Digital Rights Management and Information Dynamics.

Heon Y. Yeom received his B.S. degree in computer science from Seoul National University, Seoul, Korea in 1984 and the M.S. and Ph.D. degrees in computer science from Texas A&M University, College Station, in 1986 and 1992, respectively. From 1986 to 1990, he was with Texas Transportation Institute as a Systems Analyst and from 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University, in 1993, where he currently is a Professor and teaches and conducts research on distributed systems, multimedia systems, and transaction processing.

* The content of this paper has been partly presented at the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10) Poster session, Vancouver, BC dated on Oct/04/2010 with poster title "Backup Metadata As Data: DPC-Tolerance to Commodity File Systems."