Storage Efficient Backup of Virtual Machine Images

Artur Huletski
St. Petersburg Academic University
St. Petersburg, Russia
hatless.fox@gmail.com

***Abstract*—In spite of constant decrease of storage price per megabyte, growth of various services that provide virtual servers and administrate them makes problem of efficient storage usage still actual. Moreover such services have to backup content of virtual servers to be fault tolerant. This paper describes an implementation of block-based backup approach that uses analysis of internal file system structures to minimize required storage. The approach is based on saving only blocks with data that is actually used by a file system. The main purpose of the paper is to provide a description of how this can be done for Ext3, Ext4 and NTFS file systems and list features that should be taken into account. Proof of concept library that doesn't depend on virtual machine vendor has been developed to evaluate described approaches. Besides this a sketch of incremental backup algorithm that uses information about used file system blocks to minimize IO operations and time of restore phase is presented.**

I. Introduction

A service that holds customer data have to do some actions to prevent data loss. Data backups can help with it by storing copy of data in a safe place but they must be done as frequently as possible to be effective. Unfortunately backups are not storage-free, so the service have to sacrifice reliability and perform backups not frequently enough. Services that provide virtual workstations (cloud services) are among them so the problem with entire virtual machine (VM) image backup have arisen.

There are various proprietary systems that allow image backup and deduplication, but closed sources and the lack of white papers that describe their internal implementation make us guess about how actually such software works. Examples of such systems are: Veeam Backup & Replication, Acronis Backup & Recovery, EMC Avamar, Symantec NetBackup, Thinware vBackup. These systems provide full-fledged enterprise solutions to protect data from being lost and proprietary details of implementation allows them to compete with each other. There are two high-level approaches for backups, based on choice of backup unit:

- *File-based backup*: Such approach analyzes which files/directories require backup and saves them. This is similar to what any version control system does, so customer usually can access backed up files without full disk restoring. To determine files that should be backed up software can either add agents to guest operation system (one of EMC Avamar mode according to [1]) to track changes or analyze entire file system on every backup which is less inefficient but non-invasive approach.
- *Block-based backup*: The "atom" of backup is a binary chunk (block). Every chunk is compared with its previous version and stored if some changes have been made. Implementation of this approach is much simpler than the previous one since no file system topology analysis is required. Acronis Backup & Recovery uses this approach to provide efficient deduplication as mentioned in [2].

This paper is focused on block-based backup since it preserves underlaying file system layout and much simpler to implement. Systems listed above are complex software products, so it's reasonable to choose some subsystem and analyze how it can be implemented. This paper addresses the part that determines and retrieves data that should be backed up.

One of the approach is implemented by virtualization software vendor VMware. VMware provides a feature called Changed Block Tracking (CBT) that allows to obtain a stream of changed blocks as described in [3]. This frees engineers from manual reading entire disk (that may cause live system freeze) and from determination of changed blocks. Developers can focus only on backup process itself and on safe backups storing. Another advantage of this technology is tiny backup window size, since block is pushed to "changed block stream" as soon as possible. So load on backup hardware can be distributed more uniformly. CBT feature is not free (costs minor CPU resources for bookkeeping) so it is disabled by default. Besides several usage limitations of the feature described in [3], the main disadvantage is vendor lock. Customer must use VMware virtualization solution to be able to take advantage of CBT. This feature is used (at least can be used as an option) by all enterprise backup systems listed above.

The second approach is unused block compression (UBC). This approach doesn't depend on vendor and uses parsing of file system structures to determine which blocks are actually used and handle only them. Figuratively speaking, virtual disk can be imagined as accordion, unused blocks – as air inside it and the goal is to squeeze the accordion to pack it in the smallest possible box till next music class. This reduces size of virtual disk snapshot to size of used disk space. Sections below describe how this can be done for Ext file system family and NTFS.

UBC approach is quite common and often used in software that deals with backups. For example, RMAN from Oracle uses this approach to minimize size of database files as described in [4]. Another example is partimage. This open source software performs backup for given physical disk using blocks compression if possible according to [5]. Unfortunately, this tool doesn't support Ext4 and has only experimental support for NTFS.

The main disadvantage of UBC is zeroing out unused blocks. If guest OS is compromised, malware and rootkit data can be hidden in unused parts of file system, for example, in blocks reserved for file system structures. Since every unused block is wiped out, intrusion investigation can be limited. But from the other side malware data is also deleted.

I haven't found explicit evidence of UBC usage by enterprise backup systems listed earlier. Probably this is related to zeroing out unused data, since this limits bare metal system restoring. The following conservative heuristics are used instead:

- *Zero blocks elimination*. This approach allows to ignore empty blocks during backup. If content of disk has been wiped during file system creation this given the same storage saving as UBC until some data is deleted. By default deletion doesn't imply wiping, so a block becomes unused but has non-zero content. The more such blocks FS has the less efficient this approach becomes.
- *Archiving*. Such utils as gzip or 7zip are used to eliminate duplicated content by archiving entire image. UBC is orthogonal to this operation and can be used before it to minimize amount of data that should be archiving.

It's also worth to mention that VirtualBox managers provides ability to make snapshots. As option it provides an ability to compress virtual disk, but according to [6] the approach it uses is similar to zero blocks elimination.

II. Snapshot creation via analysis of unused file system blocks

Raw content of used blocks is not enough for disk image restoring, so the following meta data should also be saved: size of block and layout of used blocks. File systems often use bitmap for the last, the same approach can be used for backups. Storing full content of used blocks may be space inefficient if data is not changed frequently. Incremental backup can be stored instead of the snapshot one to deal with it. Such backup is described in the section below.

Since UBC can be used for both physical and virtual disks, it was decided to move implementation to a separate library that determines storage areas that are actually used. The library focuses only on file system parsing and doesn't know about disk image internals (e.g. access to it). This approach is shown in Fig. 1.

An adaptor provides access to storage medium and gets information about used file system blocks that can be translated to used storage areas for further processing. Next version of library is going to handle disk partitioning (master boot record and GUID partition table schemas) to minimize adaptor development efforts. After information about used block is retrieved other subsystems of backup system is supposed to read content from used areas and create backup file with content described above.

Before particular file system description it worth to mention general analysis strategy and assumptions:

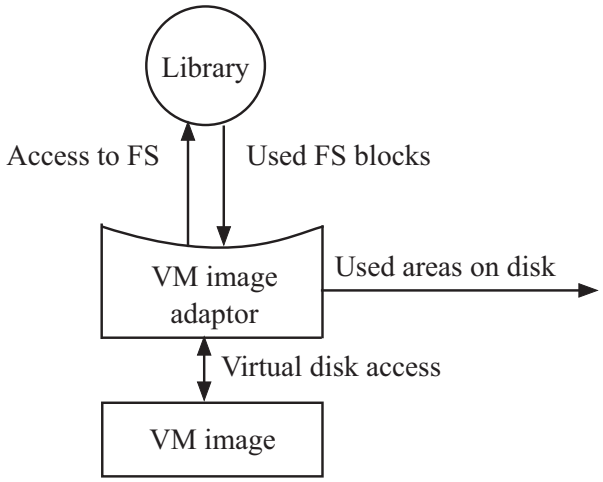

Fig. 1. Library usage scheme

- sectors that particular file system occupy on virtual disk are already known, this can be done with preliminary partition table parsing;
- file system is in consistent state, i.e. necessary state restoring has been done before analyzing;
- unsupported file system option/feature detection is logged (for further investigation and fix) and all blocks are assumed to be used, since correctness of backup is more valuable than space efficiency. This also relevant for errors of any kind (e.g. IO errors) and assertion violations (e.g. unexpected content of internal data structures' fields).

It worth to note that terms "block" and "cluster" are used for description consecutive sectors as synonyms.

A. Ext file system family

1) Common description: Ext file system has the following features that simplify UBC implementation: open sources and simple layout. The Ext file systems split all available space on *block groups* of equal size. This fact allows to minimize file fragmentation by allocation extra blocks for file content from the same group if its possible. Block usage is tracked by each group separately by treating one dedicated block inside group as a bitmap. Because of this, number blocks per block group is determined by file system block size and can be computed by multiplying block size by 8).

Block size can be retrieved from file system superblock that holds common information related to file system and is located in the begin of partition. Block group's metadata is stored in special file system structure called *group descriptor*. Number of block with block usage bitmap is stored in respective group descriptor. Every group contains copy of superblock in the first block by default as well as copies of descriptors for all groups. Fig. 2 represents common block group layout. More explicit description can be found in [7].

The main task for Ext parsing is to read all group descriptors correctly and then locate bitmap for each group. Then used blocks can be marked based on bitmap information. The

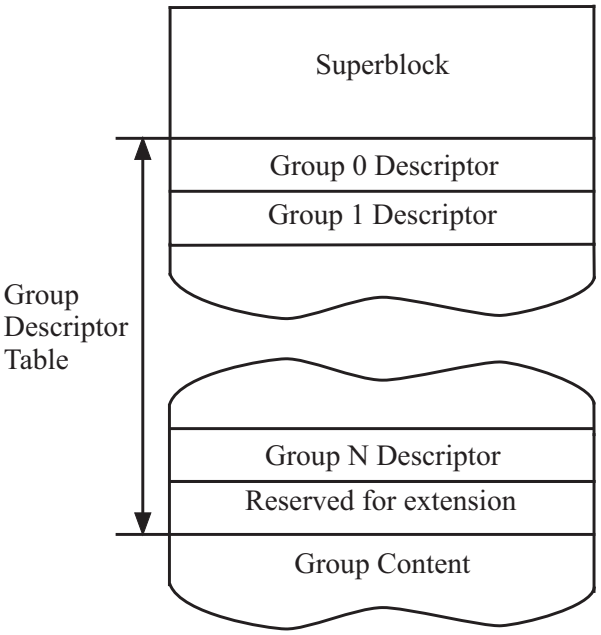

Fig. 2. Layout of Ext block group

correspondence between bits in bitmap and file system blocks is shown in Fig. 3.

Entire group descriptor table (GDT) is stored in every block group by default, right after superblock copy. So reading and parsing Ext file system is trivial by default. But there are some options/features that make life harder and require special treatment.

2) Meta group block feature: Storing entire GDT in each group is not space efficient and limits file system size. To address this issue meta block feature has been introduced.

If file system has been created with this feature enabled, extra level of data grouping called meta groups is introduced as described in [8]. As its name stands, a meta group consists of plain block groups. Descriptors of all groups in a meta group are stored in dedicated file system block. In other words GDT is split into chunks that are stored over meta groups. If no other options enabled, GDT chunk is stored after reserve copy of superblock, in the first group in meta group. So GDT is no more consecutive if the feature is enabled. High-level layout is shown in Fig. 4. Size of meta group can be determined by division file system block size on size of group descriptor. Group descriptor size can be retrieved from the superblock. Backup copies of GDT chunk is stored in the second and the last groups of a meta group.

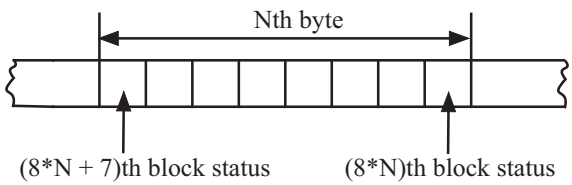

Fig. 3. Mapping between bitmap bits and blocks

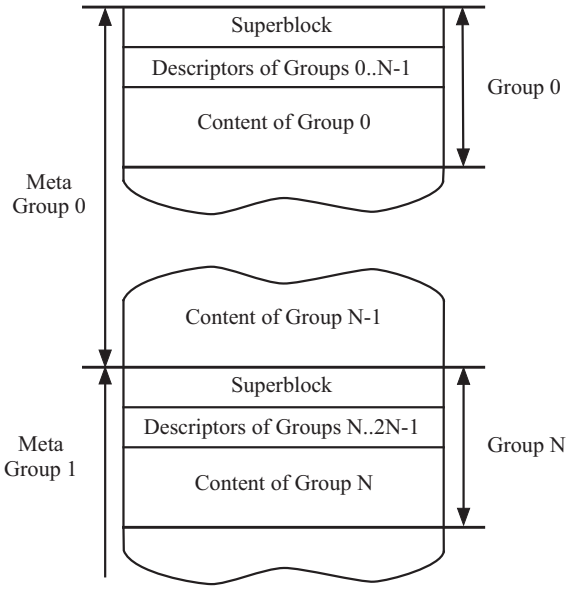

Fig. 4. Layout of Ext with meta block group feature enabled

3) 64-bit feature: The feature enables absolute block addressing using 64-bit integers instead of 32-bit ones. Unimplemented in current version of library.

4) Sparse super feature: By default every block group contains a copy of superblock in the first block. This seems a bit redundant, so sparse super feature deals with it.

The feature forces superblock copy to be stored only in groups whose index is 0 or power of 3, 5 or 7. If the feature is enabled, it affects on mapping between block number relative to group and block number relative to entire file system. For example, meta group GDT chunk is stored after reserve copy of superblock. If sparse super feature is enabled it may be stored either in the first or the second block depending on group number.

During development I found useful to define a function that translates local block number in group (local number assumes that no reserve superblock copy is stored) to absolute block number, taking into account sparse super feature by returning number of blocks occupied by superblock backup.

5) Uninit block group feature of Ext4: This feature allows to mark some block groups as uninitialized. This allows to speedup disk formatting and further consistency checks.

At first glance, looks like uninitialized groups can be safely skipped. But backup-restore tests have revealed the following: even if block group is uninitialized, it turns out that superblock reserve copy is stored in it anyway, as well as backups of meta group GDT chunks (in the second and the last groups of meta block group). So we should mark these blocks as used instead of skipping uninitialized block group.

6) Flex block group feature of Ext4: If this feature is enabled, various meta data (e.g. block group bitmap) of several consecutive groups is stored successively to speed up access to it (by better spatial locality). This feature doesn't require extra handling, since block group descriptor contains absolute

reference (block number) to a bitmap block related to particular group.

7) Summary: To retrieve used blocks from Ext to following steps should be done:

1) Read and parse superblock. This gives file system block size, size of block group, size of file system and list of enabled features.
2) Read GDT. As it was described, GDT may not be stored consecutively (meta group option enabled) on disk. In this case, GDT should be assembled manually. To minimize memory footprint and abstraction from the layout of GDT, iterator pattern can be applied. Otherwise, entire GDT can be read from the first block group.
3) For each GDT entry (descriptor) perform check whether the group has been initialized. If it wasn't mark blocks with file system related backups (super and meta group blocks) as used and process next entry. Otherwise, read address of block that contains group bitmap from descriptor and add it to used blocks information.

B. NTFS

1) Common description: The main problem of NTFS parser development I've faced is absence of source code and open specification. Some sort of official description can be found in [9]. Carrier [7] provides detailed description of NTFS internal structures and high-level algorithm for their analysis. Concise information with several good examples made by Russon and Fledel [10] (part of Linux-NTFS Project). NTFS has been designed to be robust and flexible but sometimes such design leads to circularities in file system read logic that are discussed below. Unfortunately listed sources don't help much to solve them.

Every kind of data is stored in files by design. Every file has a descriptor that is stored in special Master File Table (MFT). Needless to say that MFT content itself stored in a special file called $MFT. Even superblock is stored in file ($Boot). This is the only file that stores its content by fixed offset (content is stored in the begin of partition). To determine blocks that store content of other files, appropriate entries of MFT should be parsed.

Core system NTFS files have "well-known" indexes in MFT table. For example, $MTF file that stores MFT content occupies an entry with zero index (the first entry in MFT). Global number of the first $MTF block is stored in file system superblock. Superblock is stored in the first partition block (file $Boot).

Fortunately, NTFS uses special file to track block usage. This file is called $Bitmap and described by the 6th MFT entry. Each bit of its content corresponds to some file system block and is set to one is block is used. Mapping between bits and blocks matches the scheme used by Ext file system (Fig. 3).

2) File content extraction: MTF entry has quite simple format: it has entry header (describes common information, such as signature, used status, etc.) and an array of attributes that describe various data related to file. Examples of such attributes and entry header format is described in [7].

Every attribute has header that contains an attribute type (defines what information is stored) and a flag that indicates how information is stored.

There are two approaches to store attribute data:

- *resident* - stores data inside of MFT entry in attribute body. Offset and size are defined in header;
- *non-resident* - stores data somewhere in partition. Attribute body holds meta data called dataruns that describes actual location. Used since MFT entry has fixed size.

Attributes that contain data which size is close or more that MTT entry size are typically non-resident. Information about such external (with respect to MFT entry) storage is defined in dataruns. Datarun array defines mapping between Virtual Cluster Number (cluster offset in specific file) to Logical Cluster Number (cluster cluster offset in entire file system). Each datarun entry defines chunk of consequent clusters by storing start LCN and chunk's length, Fig. 5 shows datarun layout. Since chunks should appear to be sequential in file, VCN numbering is implicitly defined. Start and end VCN are defined in attribute header. Examples of datarun usage can be found in [10].

File content is stored in $Data attribute that has $0x80$ type by default. NTFS has alternate data streams feature that allows to define several (named) file contents for a single file. The "main" data stream has $Data attribute with empty name, so additional check of name length is required.

I found useful to implement access to NTFS file content in a way similar to standard C library instead of direct dataruns manipulation. This separates parser logic and internal organization of NTFS file content.

3) Sparse and compressed files: NTFS uses two common approaches to save storage space: sparse and compressed files.

Sparse files allow to store files with empty blocks without actual cluster allocation until it is necessary. This is achieved by setting LCN field size to zero in data run. Such sparse content should be treated as zeros.

Compressed files are described in [7] and [10]. They are not supported by library for now since I'm not sure whether compression is used for $MFT and $Backup system files. Since both this files are read very often, making them compressed would likely cause performance degradation of read/write operations.

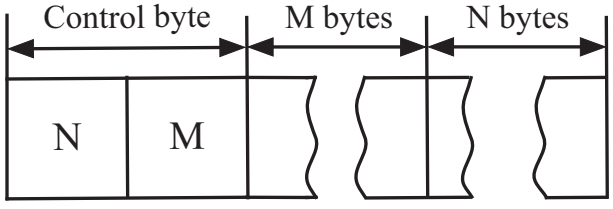

Fig. 5. Layout of NTFS datarun

4) Fixup values: NTFS slightly modifies sectors that hold MTF entry to add additional level of sector failure detection. The last two bytes of every such sector are replaced with update sequence number (USN). USN is an index of last entry in NTFS change journal related to file described by entry. This should be taken into account during MFT entry reading, original content of sectors is stored in MFT entry header according to [10].

5) Bad clusters: Information about bad clusters detected by file system is stored in special file called $BadClus (the 8th MFT entry). Filtering bad clusters from used ones can be done as additional optimization. This is not implemented in the library.

6) Attributes types definition: NTFS provides ability to define custom attributes and to change types of existent ones. Description of attributes is stored in $AttrDef file (the 4th entry). So type of $Data attribute in theory may differ from default one. Look like this is not relevant to $Data attribute of $MFT file since to know data type of $Data attribute we have to read $MFT content first. Handling of changed $Data type hasn't been added to library by now but its implementation is straightforward and is going to be added in future.

7) Multientry records: Since there is no limit to number of attributes for each file, situation when there is no enough space in MTF entry for all attribute headers is possible. Additional MFT entries are allocated to store attributes to handle this. Such entries and attributes they stored are described with attribute of type $ATTRIBUTE_LIST. This attribute is stored in base MFT entry and contains a list of all attributes with index of MTF entry where they are stored.

So, in theory $MTF file may occupy several MFT entries and file system usage pattern that makes $MFT extremely fragmented can be created in theory by storing huge file, than small file to fill up the disk (small file will be stored in MFT zone), than remove the huge file and repeat process.

Since dataruns can be located in several $Data attributes, there can be either too many $Data attributes to store in a single MFT entry. So $ATTRIBUTE_LIST should be used to add extra MFT entry. But this leads us to loop in reading logic: to read $MFT content we have to read some MFT entries stored in $MFT content.

For our task this may not be relevant, since $Bitmap is likely stored near the base MTF entry and can be found by analyzing near blocks. But $Bitmap itself can be possibly fragmented (in theory) by continuous file system partition increasing and filling it up.

Source code of Ntfs.sys would probably demystify this assumptions and looks like disassembling is ultima ratio. Since described cases are quite unusual, theoretical and probably can be "fixed" with defragmentation, multientry file records are not handled by the library.

8) Summary: High-level strategy for used blocks extraction is described in Algorithm 1.

Variables have the following meaning:

- *super_data* – super block structure;
- *mft_entry*, *bitmap_entry* – entries from MFT that corresponds to $MFT and $BITMAP files respectively;
- *mft_content* – content of $MFT file (MFT table);
- *block_usage_data* – bitmap of used blocks of entire filesystem;
- get_base_mft_entry, get_mft_entry – both functions return required MFT entry. The first one returns entry with zero index ($MFT file) using information from file system super block, the second one returns entry with requirud index. Note, that fix-up values must be replaced with original content of MFT entry as it have been described;
- constant *BITMAP_IND* – index of $BITAMP file in MFT, has value 6;
- get_file_content – returns entire file content. To reduce memory footprint, abstraction that allows to read the content can be returned. Content reading requires searching for $DATA attribute and proper data runs handling which was described above;

Algorithm 1 NTFS used blocks retrieval

```
super_data = read_and_parse_superblock();
mft_entry = get_base_mft_entry(super_data);
mft_content = get_file_content(mft_entry);
bitmap_entry = get_mft_entry(mft_content, BITMAP_IND);
block_usage_data = get_file_content(bitmap_entry);
```

III. Implementation of incremental backups based on used blocks information

Snapshot backup is a backup that contains enough information to restore disk content from scratch. Incremental backups contain only data that has been changed during time passed from previous backup. Such backups are used to save storage space in cost of restore time. Strategy that mimics approach used by version control systems (VCS) (e.g. Mercurial as described in [11]) can be used to choose type of backup that should be made. Backups are stored as incremental while storage cost of all incremental backups made before a snapshot backup is less than new snapshot backup. The file system is isomorphic to plain text file in terms of backup by mapping blocks to file lines.

Given approach assumes that file system has constant size which is the most common case. This allows to stop restore process as soon as possible. The algorithm can be modified with tracking auxiliary info about previous file system sizes to support partition changes.

A. Formats of snapshot and incremental backups

To represent a snapshot we need to store the following:

- *Common file system data* such as block size and total number of blocks.
- *Bitmap* of used blocks to save file system layout.
- *Content of used blocks*.

To represent incremental backups we will use the following data:

- *Common file system data* like in snapshot backup.
- *Bitmap* for blocks that have changed their content or become unused.
- *Bitmap* of blocks that have become unused. Can be represented via array if this is more space efficient.
- *Content of changed blocks*.

Snapshot backup consists of file system info, bitmap of used blocks and content of used blocks. When backup should be made snapshot backup is created first. If its reasonable to convert it to incremental one, the conversion is performed. Backup software can store last taken snapshot backup to perform such conversion.

An incremental backup can be created using Algorithm 2, assuming common bitwise operations are implemented for bitmaps.

Algorithm 2 Incremental backup creation

```
changed_blocks_bm = diff_blocks(base, new, new.bm);
inc.unused = base.bm and (not new.bm);
inc.ch_bm = changed_blocks_bm or inc.unused;
inc.data = extract_blocks(new, changed_blocks_bm);
base = new
```

Operations and variables description:

- *changed_blocks_bm* – bitmap with marked changed blocks;
- diff_blocks – goes through bitmap given in 3rd param, and compares blocks from base snapshot and new snapshot. Returns bitmap where different blocks are marked with ones;
- extract_blocks – returns blocks that are marked with given bitmap;
- *base* – last created snapshot backup;
- *new* – recently created snapshot backup;
- *new.bm* – bitmap of recently created snapshot backup;
- *inc* – created incremental backup;
- *inc.unused* – bitmap with unused blocks of created incremental backup;
- *inc.ch_bm* – bitmap of changed and used blocks of incremental backup;
- *inc.data* – content of changed blocks of incremental backup;

B. Restore snapshot by incremental updates

The naive approach is to apply incremental updates one by one to latest snapshot in order of creation. But more efficient strategy can be developed based on fixed file system size knowledge. Instead of going through all states that file system had in past, lets try to assemble the state we want to restore by going through incremental backups in opposite direction. Note that entire file system partition is assembled, not a snapshot.

Lets start from assumption that all blocks were changed and create a bitmap filled with ones. It defines blocks we want to restore. Then go through backups as if they have been put in a stack during creation. There are to possible cases:

- *Incremental backup*.
  First, perform bitwise *and* for bitmap with blocks we want to restore and inverted bitmap of changed blocks from current incremental backup. This gives us bitmap with blocks that are uninitialized after current backup handling. All changed blocks (except unused ones) from current backup should be written to result image. The last operation can be performed by other thread. Since content of determined changed block is the most actual, any previous changes in a block can be safely ignored. If bitmap of blocks that should be restored is empty – backup can be treated as completed.
  This is correct since no more blocks that require update and no extra blocks can appear from previous changes.
- *Snapshot backup*. In this case both bitmaps are scanned simultaneously and all blocks that require to be set are copied to result file system image.

This approach is described by Algorithm 3 in more formal way.

Algorithm 3 Disk image restore

```
uninit_bm = init_bm(fs_size, 1);
disk_image = init_disk_image(raw_size);
backup = get_last_backup();
while !bm_is_empty(uninit_bm) do
  if is_incremental(backup) then
    uninit_bm = uninit_bm and (not backup.ch_bm);
    update_blocks(disk_image, backup);
  end if
  if is_snapshot(backup) then
    snapshot_update(disk_image, uninit_bm, backup);
    clear_bm(uninit_bm);
  end if
  backup = get_next_backup(backup);
end while
```

Variables and functions description:

- *uninit_bm* – bitmap with uninitialized blocks marked;
- *fs_size* – size of restored file system in blocks;
- init_bm – creates bitmap of given size, default state is passed as the second parameter;
- *raw_size* – size of file system in bytes;
- *disk_image* – image (file) with restored file system;
- init_disk_image – creates image of given size initialized with zeroes;
- *backup* – backup to be handled;
- *backup.ch_bm* – changed and unused bitmap of current backup;

- get_last_backup – retrieves the most recent backup made for restored image;
- *bm_is_empty* – checks whether all items are not marked in given backup;
- is_incremental – checks whether given backup is incremental;
- update_blocks – copies content of blocks that marked as changed by given incremenetal backup;
- is_snapshot – checks whether given backup is snapshot;
- snapshot_update – copies content of required blocks (marked by the seconds parameter) from given snapshot backup;
- clear_bm – clears given bitmap.

This approach allows to individual block updates to be independent and be performed only once, so this process can be done concurrently. Main thread can determine which blocks should be stored and issue a new task for image block initialization, so fork-join framework described in [12] can be used naturally.

IV. Performance and evaluation

Since market solutions have closed sources and there is no precise description of how snapshot compression in performed the only available strategy is to use such software as a black box. By the time of writing only library and test environment is implemented (without adaptors), so only expected evaluation can be done.

UBC approach allows to store exactly actual data used by filesystem, so degree of efficiency (comparing with plain copy) linearly depends on disk usage percentage. Further processing related to particular filesystem structure can reduce this amount by ignoring blocks that contain file systems backups (super blocks, GDT chunks for Ext).

Current implementation of Ext analysis costs O(n) time and O(n) memory, where n is size of image. Both bounds are related to GDT size (depends on image size and size of FS block). Memory usage can be reduces to O(k) (k – size of file system block) by replacing entire GDT loading with iterative approach.

NTFS analysis costs O(m) time (m – number of dataruns in $BITMAP file) and O(k) memory (k – size of filesystem block).

V. Conclusion

Virtual disk image compression based on information about used blocks introduces simple and universal way to save storage space, since it doesn't take into account actual file content and doesn't require deep parsing of underlying file system (in case of described Ext* and NTFS). Library that allows to retrieve usage info from listed file system has been implemented to verify this approach. Algorithm that uses incremental backups and by design minimizes IO by performing incremental backups in reverse order has also been described.

The main drawback of unused block compression is not being deduplication friendly, since we can operate only with binary data. Natural solution is to create huge hash map and store only indexes instead of actual blocks (as mentioned in [2] and [13]) but it can be extremely inefficient and requires further investigation.

Another drawback is full disk read in the worst case to create snapshot backup.

References

[1] EMC, "EMC Avamar backup and recovery for VMware environments", unpublished white paper.

[2] Acronis, "How Deduplication Benefits Companies of All Sizes", unpublished white paper.

[3] VMware knowledge base website, Changed Block Tracking (CBT) on virtual machines (1020128), Web: http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1020128.

[4] Oracle online documentation website, 8 RMAN Backup Concepts, Web: http://docs.oracle.com/cd/E11882_01/backup.112/e10642/rcmcncpt.htm#BRADV002.

[5] Partimage website, Main page, Web: http://www.partimage.org/Main_Page

[6] VirtualBox online website, 8.23 VBoxManage modifyhd, Web: http://www.virtualbox.org/manual/ch08.html#vboxmanage-modifyvdi

[7] B.Carrier, *File System Forensic Analysis*. Upper Saddle River, NJ: Pearson Education, 2005.

[8] Ext4 Wiki website, Ext4 Disk Layout, Web: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[9] Microsoft TechNet website, How NTFS Works, Web: http://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx.

[10] R.Russon, Y.Fledel, "NTFS documentation", unpublished.

[11] B.O'Sullivan, *Mercurial: the Definitive Guide*. Sebastopol, CA: O'Reilly Media, 2009.

[12] D.Lea, "A Java fork/join framework", *in Proc. ACM 2000 Conf.*, 2000, pp. 36-43.

[13] S. Quinlan, S. Dorward, "Venti: a new approach to archival storage", *in Proc. FAST 2002 Conf.*, 2002, Article No. 7.