TECHNISCHE FAKULTÄT

Department Informatik

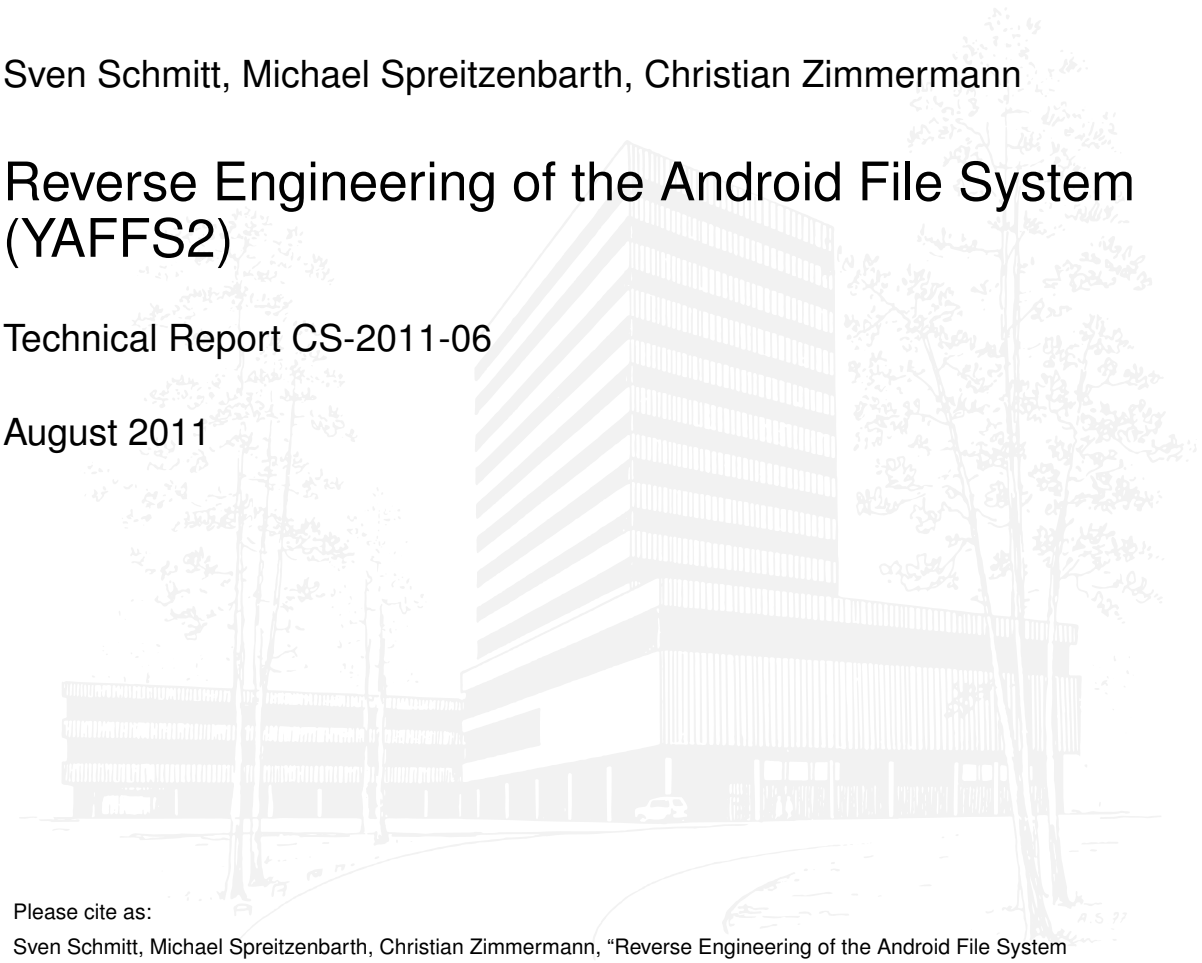
Technical Reports / ISSN 2191-5008

Sven Schmitt, Michael Spreitzenbarth, Christian Zimmermann

Reverse Engineering of the Android File System (YAFFS2)

Technical Report CS-2011-06

August 2011



Please cite as:
Sven Schmitt, Michael Spreitzenbarth, Christian Zimmermann, “Reverse Engineering of the Android File System (YAFFS2),” University of Erlangen, Dept. of Computer Science, Technical Reports, CS-2011-06, August 2011.

Friedrich-Alexander-Universität Erlangen-Nürnberg

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.informatik.uni-erlangen.de

Reverse Engineering of the Android File System (YAFFS2)

Sven Schmitt, Michael Spreitzenbarth, Christian Zimmermann
Security Research Group
Dept. of Computer Science, University of Erlangen, Germany
www1.informatik.uni-erlangen.de

***Abstract*—YAFFS2 is a file system which is used in many modern smartphones. Allthough YAFFS2 is an open standard and there exists an open source implementation, the behavior of YAFFS2 is not very well understood. Additionally, several aspects like wear-leveling and garbage-collection are not well-specified in the standard so that their actual behavior has to be reverse engineered from the implementation. Here, we give an introduction to and describe the basic functionality of YAFFS2. We place a particular focus on the detailed analysis of both wear-leveling and garbage-collection mechanisms, since these are important within a forensic analysis of the file system.**

***Index Terms*—smartphones; forensics; Android; YAFFS2, garbage-collection**

I. Introduction

As smartphones running the Android platform become more and more popular, there is also an increasing need for methods and tools to forensically analyze digital data that is stored within such devices. Therefore, it is necessary to understand the characteristic structures and mechanisms of the underlying file system that actually handles the internal flash memory. One of the file systems used by the Android platform is YAFFS2 (Yet Another Flash File System 2). YAFFS2 is particularly interesting because of its file-system level implementation of *wear-leveling*, a concept to prevent premature wear-out of flash memory. Even though the Android platform makes use of the EXT4 file system beginning in 2011 with version *Gingerbread* (Android 2.3), there are still many devices in use running a lower version than 2.3 and thus are using YAFFS2. Therefore, insights into the amount and quality of evidence left on YAFFS2 devices is still of major interest.

In this report, we give an introduction to YAFFS2 and its basic design principles. As YAFFS2 is an open standard [1] and an open source implementation exists [2], we originally thought that it would be easy to understand the behavior of the file system. However, since the source code is subject to constant development and the standard does not give details to many relevant operations (especially wear-leveling and garbage-collection), we had to indulge in reverse engineering activities to understand its behavior.

Within this report, we analyzed version 0bc94484426d0aa0db445360d6e5845696936229 of the YAFFS2 source code dated June 2011 [3]. We give many excepts of the file system's source code that allows for a better understanding of its characteristic mechanisms, such as garbage collection and its own implementation of wear-leveling, which is normally implemented on the controller of the used memory chip. The main goals of this report are to give an in-depth understanding about YAFFS2 and to reveal discrepancies between its specification and actual behavior with respect to forensic analysis.

This report is outlined as follows: In Section II, we introduce YAFFS2 and its basic design. In Section III, we analyze techniques of the file system to manage data storage. This also includes an analysis of it's actual behavior regarding data organization and a comparison of this behavior to the one described in the specification. As YAFFS2's garbage collection techniques are likely to have a major impact on the amount of data that can be restored from a storage device, we provide an in-depth analysis of YAFFS2's garbage collection techniques in Section IV. Additionally, in Section V, we analyze YAFFS2's techniques to minimize wear of NAND flash memory devices.

II. YAFFS2 basics

YAFFS2 is a file system designed mainly by Charles Manning, especially for use with NAND flash memory devices. It is the successor to YAFFS1 and shares some basic characteristics with its predecessor. However, nowadays, the use of YAFFS1 is deprecated, because it violates the specification of some NAND flash memory

chips by writing deletion markers into the Out-Of-Band (OOB) area of pages that contain contents of deleted objects. The violation is, that the OOB area of pages are rewritten directly without erasing the respective blocks beforehand. Additionally, YAFFS1 can only handle NAND flash memory pages with a maximum size of 512 bytes, while YAFFS2 supports page sizes of 2048 bytes and more [4].

To correspond to the layout of NAND flash memory, YAFFS2 structures data in so-called *chunks* and blocks, with a chunk being the unit of allocation and writing and a block being the unit of erasure. To write data to a NAND flash memory device, YAFFS2 first allocates a block and then writes chunk-wise within this block. Typically, the size of a chunk equals the size of a page on the NAND flash memory, but, if necessary, a chunk can also be mapped to several pages, for example to write to several NAND flash memories in parallel [4]. In the following, unless stated otherwise, a chunk has the size of a page and the terms chunk and page are used synonymously. For simplification, the terms *NAND flash memory device* and *NAND* are also used synonymously.

To YAFFS2, anything that can be stored in the file system is, first of all, an *object*. The way YAFFS2 manages a specific object depends on the type of the object. An object can either be a data file, a directory, a hard link, a soft link or a special object, such as a device file or a pipe. During mounting of a YAFFS2 device, information about all objects on the device, as well as information about the device itself are loaded into RAM. This includes building a directory structure in RAM to be able to find objects by name, as well as building a hash table to be able to find objects by their unique object number. To speed up mounting of a device, YAFFS2 stores parts of this RAM structure in specially reserved blocks on the device. This so called *checkpoint* contains information about the device, the state of its blocks and information about objects and their chunks on the device including the directory structure. The number of blocks reserved for this checkpoint and the number of these blocks in use are relevant to the intensity with which YAFFS2 tries to perform garbage collection. More information about YAFFS2's garbage collector is provided in Section IV.

To be able to allocate blocks, identify bad blocks and select blocks for delete operations, the file system distinguishes between ten states a block can be in. These states are defined in lines 229 to 273 of file `yaffs_guts.h`. During runtime, YAFFS2 keeps state information of all blocks in RAM. As can be seen in Figure 1, blocks can only transition from one state into another in a predefined order.

On a device that has never been mounted using YAFFS2 or if YAFFS2 cannot recover the block states from a checkpoint, all blocks are in initial state `UNKNOWN`. In that case, a scan of the device is necessary to determine the state of all blocks. The scan is performed as defined in function `yaffs2_ScanBackwards` in lines 911 to 1539 of file `yaffs_yaffs2.c`. If a block turns out to be unusable due to production errors or wear out, its state transitions from `UNKOWN` into `DEAD`. If a block is not unusable it needs scanning to determine its state and transitions from state `UNKNOWN` into state `NEEDS SCANNING` until it has actually been scanned. While being scanned, a block is in state `SCANNING`. During runtime, a block can either be in state `FULL`, `EMPTY`, `ALLOCATING`, `DIRTY` or `COLLECTING`. The block in state `ALLOCATING` is the block currently selected for write operations. A block stays in state `ALLOCATING` until it is completely filled. A block is in state `FULL` if all of its chunks have previously been allocated and at least one chunk contains current data. A block that has not been fully allocated and is not currently selected for write operations is also in state `FULL`. This can happen, if a device is disconnected from its power source without proper unmounting. In this case, the block's state is set to `FULL` on scanning after the device is mounted again. A block in state `EMPTY` has been erased and does not contain any data. An empty block can be allocated for regular write operations or to store checkpoint data. Full blocks are examined by the garbage collector to check whether they contain current chunks. If the garbage collector becomes active and copies current chunks from a full block to the block in state `ALLOCATING`, the block that is being copied off is in state `COLLECTING`. A block that contains only obsolete chunks is in state `DIRTY` and can transition into state `EMPTY` by being erased. YAFFS2 always reserves some blocks on a device to store checkpoint data. A block containing checkpoint data is in state `CHECKPOINT`.

When allocating blocks and chunks, YAFFS2 follows two main principles to match modern NAND flash memories' specifications. These are a *zero overwrite* policy and sequential writing of chunks within a block. Although it is technically possible to change bits of an already written page from a logical one to a logical zero without having to erase the whole block containing the page, many modern NAND flash memories' specifications deprecate such rewrites to improve flash memories'

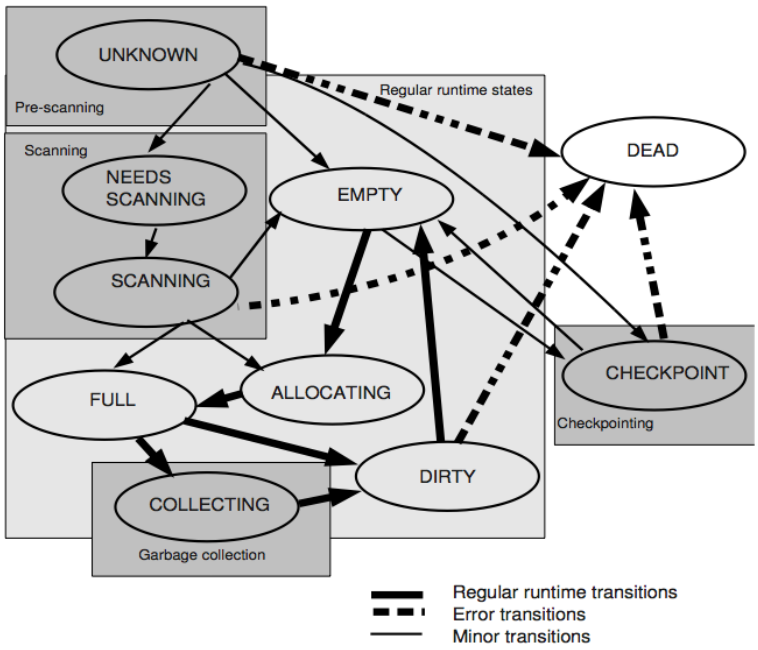


Fig. 1. YAFFS2's block states [4]

reliability and lifespan. YAFFS1 violates these specifications by overwriting a byte in the OOB area of a page to mark the page's contents as deleted. YAFFS2 refrains from writing these deletion markers and never overwrites already written chunks without erasing their blocks beforehand, hence following a zero overwrite policy. Typically, modern NAND flash memories' specifications also require sequential writing of pages within a block. As depicted in Listing 1, YAFFS2 complies to this requirement by allocating new chunks in sequential order within a block.

Strictly sequential writing of chunks within a block does not only comply to NAND flash memory specifications but also enables YAFFS2 to keep a chronological log of file modifications. Additionally, as can be seen in Listing 2, as long as empty blocks can be found on a NAND, YAFFS2 also tries to allocate blocks for writing in sequential order. These methods of allocating chunks and blocks make YAFFS2 a truly log-structured file system. The actual techniques to organize the log are introduced and analyzed in Section III. As the file system is a log-structured and follows a zero overwrite policy, it is inevitable that a certain amount of obsolete data is almost always stored somewhere on the NAND. To identify and delete this obsolete data, a garbage collector is needed. We provide an analysis of the garbage collection mechanism in Section IV.

III. Data organization and OOB area tags

In the following, we analyze the techniques used by YAFFS2 to manage data and organize its log. In Section III-A, we introduce YAFFS2's OOB area tags. YAFFS2's way to perform modifications and deletions of objects are then introduced in Section III-C. When data has to be written to or read from a NAND, YAFFS2 distinguishes between data chunks and header chunks. While data chunks contain actual content of an object, header chunks are used to identify the object's type and to store meta data about the object, such as its name, size and timestamps. Data chunks are only used to store file contents whereas directories and links only consist of header chunks.

A. OOB area tags

In order to be able to associate chunks with an object and store information such as a chunk's position within an object, the OOB areas of NAND flash memory's pages are used. The OOB areas are also used to chronologically organize YAFFS2's log. For that purpose, a sequence number is used. As can be seen in lines 1982 and 1983 of Listing 2, every time a block on a device is allocated for writing, the device's block sequence number is incremented. Every chunk that is written to the newly allocated block is marked with the current block sequence number within the chunk's OOB area. That

```
2019 static int yaffs_AllocateChunk(yaffs_Device *dev, int useReserve,
2020                 yaffs_BlockInfo **blockUsedPtr)
2021 {
2022         int retVal;
2023         yaffs_BlockInfo *bi;
2024
2025         if (dev->allocationBlock < 0) {
2026         /* Get next block to allocate off */
2027         dev->allocationBlock = yaffs_FindBlockForAllocation(dev);
2028         dev->allocationPage = 0;
2029         }
2030         [...]


2041         /* Next page please.... */
2042         if (dev->allocationBlock >= 0) {
2043                 bi = yaffs_GetBlockInfo(dev, dev->allocationBlock);
2044
2045                 retVal = (dev->allocationBlock * dev->param.nChunksPerBlock) +
2046                         dev->allocationPage;
2047                 bi->pagesInUse++;
2048                 yaffs_SetChunkBit(dev, dev->allocationBlock,
2049                                 dev->allocationPage);
2050
2051                 dev->allocationPage++;
2052
2053                 dev->nFreeChunks--;
2054
2055                 /* If the block is full set the state to full */
2056                 if (dev->allocationPage >= dev->param.nChunksPerBlock) {
2057                         bi->blockState = YAFFS_BLOCK_STATE_FULL;
2058                         dev->allocationBlock = -1;
2059                 }
2060
2061                 if (blockUsedPtr)
2062                         *blockUsedPtr = bi;
2063
2064                 return retVal;
2065         }
2066
2067         T(YAFFS_TRACE_ERROR,
2068                 (TSTR("!!!!!!!!! Allocator out !!!!!!!!!!!!!!!!!" TENDSTR)));
2069
2070         return -1;
2071 }
```

Listing 1. Function `yaffs_AllocateChunk` (Excerpt from yaffs_guts.c)

way, the block that has been written to most recently has always the highest block sequence number. Thus, a chronological order of blocks can be derived from their sequence numbers, regardless of the blocks' physical position on the device. As chunks are allocated strictly sequentially within a block, starting from the lowest address of the block, all chunks within a block are in chronological order by default.

Along with the block sequence number, several other important values are written to a page's OOB area. All OOB area tags used by YAFFS2 can be seen in Table I. Other meta data such as timestamps are not written to a header chunk's OOB area, but into the data area of the chunk.

During our analysis of YAFFS2's actual behavior, it became obvious that, although all tags shown in Table I were actually written to the OOB areas, the actual size of the `nByte` tag did not necessarily always match the size stated in Table I. The reasons for this discrepancy and the actual meanings of YAFFS2's OOB tags are provided below. It is important to know, that the actual order in which all aforementioned tags are written to the OOB areas of a NAND is not controlled by YAFFS2 itself but by the NAND flash memory driver used by the operating system to access the flash memory chip [5]. Because of that, it is difficult to perform a fully automated analysis of a NAND dump. In the following we will discuss the different area tags in detail.

1) blockState: The `blockState` tag is used to mark bad blocks by writing a value different than `0xff` to the

```
1954 static int yaffs_FindBlockForAllocation(yaffs_Device *dev)
1955 {
1956         int i;
1957         yaffs_BlockInfo *bi;
1958         [...]

1969         /* Find an empty block. */
1970
1971         for (i = dev->internalStartBlock; i <= dev->internalEndBlock; i++) {
1972                 dev->allocationBlockFinder++;
1973                 if (dev->allocationBlockFinder < dev->internalStartBlock
1974                     || dev->allocationBlockFinder > dev->internalEndBlock) {
1975                         dev->allocationBlockFinder = dev->internalStartBlock;
1976                 }
1977
1978                 bi = yaffs_GetBlockInfo(dev, dev->allocationBlockFinder);
1979
1980                 if (bi->blockState == YAFFS_BLOCK_STATE_EMPTY) {
1981                         bi->blockState = YAFFS_BLOCK_STATE_ALLOCATING;
1982                         dev->sequenceNumber++;
1983                         bi->sequenceNumber = dev->sequenceNumber;
1984                         dev->nErasedBlocks--;
1985                         T(YAFFS_TRACE_ALLOCATE,
1986                           (TSTR("Allocated block %d, seq  %d, %d left" TENDSTR),
1987                            dev->allocationBlockFinder, dev->sequenceNumber,
1988                            dev->nErasedBlocks));
1989                         return dev->allocationBlockFinder;
1990                 }
1991         }
1992
1993         T(YAFFS_TRACE_ALWAYS,
1994           (TSTR
1995            ("yaffs tragedy: no more erased blocks, but there should have been %d"
1996             TENDSTR), dev->nErasedBlocks));
1997
1998         return -1;
1999 }
```

Listing 2. Function `yaffs_FindBlockForAllocation` (Excerpt from yaffs_guts.c)

Field	Size for 1024 bytes chunks	Size for 2048 bytes chunks
`blockState`	1 byte	1 byte
`chunkID`	4 bytes	4 bytes
`objectID`	4 bytes	4 bytes
`nBytes`	2 bytes	2 bytes
`blockSequence`	4 bytes	4 bytes
`tagsEcc`	3 bytes	3 bytes
`ecc`	12 bytes	24 bytes

TABLE I
YAFFS2 OOB AREA TAGS ACCORDING TO YAFFS2'S OFFICIAL SPECIFICATION [1]

respective byte within a bad block's pages' OOB areas.

2) objectID: The `objectID` tag contains the unique object number of the object a chunk is associated with. As depicted in Listing 3, certain values are reserved for special *pseudo* objects, such as the root directory of a device or the `lost+found` folder. The `lost+found` directory is not written to a NAND. Instead, it is created in RAM on mounting of the NAND.

The `unlinked` and `deleted` objects are used by YAFFS2's to perform object deletions. Their meaning is provided in Section III-C3. All non-pseudo objects have an `objectID` tag value of at least 257. In Listing 4, the procedure to assign an object number to a newly created object is depicted, showing that no object numbers below 257 can be assigned to objects other than the aforementioned pseudo objects. This is because

```
78 [...]
79 /* Some special object ids for pseudo objects */
80 #define YAFFS_OBJECTID_ROOT             1
81 #define YAFFS_OBJECTID_LOSTNFOUND       2
82 #define YAFFS_OBJECTID_UNLINKED         3
83 #define YAFFS_OBJECTID_DELETED          4
84 [...]
```

Listing 3. Possible values for the `objectID` tag (Excerpt from yaffs_guts.h)

the variable `bucketFinder` is of type `unsigned integer` (see line 693 of `yaffs_guts.h`) and thus has a minimal value of zero. As the variable `bucketFinder` is incremented at least once inside of function `yaffs_FindNiceObjectBucket`, the variable `bucket` in line 1365 of Listing 4 has an initial value of at least one. Thus, as `YAFFS_NOBJECT_BUCKETS` has a value of 256 (see line 61 of `yaffs_guts.h`), the lowest object number available to a non-pseudo object is 257.

In a data chunk, the `objectID` tag contains just the object number of the object the data chunk is associated with. However, in a header chunk, the `objectID` is not only used to associate the chunk with an object, but also to identify the type of the object. As can be seen in line 83 and 84 of Listing 5, the highest byte of the `objectID` field is used for that purpose. In YAFFS2's documentation, the extended use of the `objectID` tag is mentioned briefly, but no indication of YAFFS2 using extended tags by default is given. During our analysis we detected the values depicted in Table II being used in the `objectID` field's highest byte. For example, a file with object number 300 has `0x1000012c` as value for its header chunk's `objectID` tag, whereas a directory with object number 300 would have `0x3000012c` as value for its header chunk's `objectID` tag. The object type is also written to Byte 0 of the header chunk's data area. That way, YAFFS2 is able to identify the type of an object, regardless of extended tags being used or not.

3) chunkID: Basically, the `chunkID` tag defines a chunk's position within an object. In a data chunk, the value of the `chunkID` tag defines the chunk's offset from the beginning of an object, e.g. the data chunk with a `chunkID` tag value of four is the fourth data chunk of its object and the data chunk with a `chunkID` tag value of one is the first data chunk of its object. Regarding header chunks, YAFFS2's documentation claims that *a chunkId of zero signifies that this chunk contains an objectHeader* [4]. However, during our analysis, no chunks with a `chunkID` tag value of zero could be found. Instead, we observed, that the value of a header chunk's `chunkID` tag is constituted of two parts, namely an additional flag in the top byte of the `chunkID` tag and the object number of the object's parent directory in the other three bytes of the `chunkID` tag. As can be seen in Listing 5 and Listing 6, the reason for that is, that YAFFS2 uses its extended tags functionality. In doing so, YAFFS2 writes at least `0x80` to the top byte of a header chunk's `chunkID` tag and the the object number of the the header chunk's object's parent directory to the remaining three bytes of the `chunkID` tag. The object number of an object's parent directory is also stored in Bytes 4 to 7 of the object's header chunk's data area to enable YAFFS2 to identify the object's parent directory regardless of extended tags being used or not.

If a file's header chunk is marked with a *shrink header marker* to indicate a hole inside the file (see Section III-A7), the `chunkID` tag's top byte is set to `0xC0` by an OR-operation of `0x80` and `0x40`. YAFFS2 uses its extended tag functionality to speed up scanning of a device by writing more information into tags than absolutely necessary. Although writing the value zero into the `chunkID` tag would be enough to identify a header chunk, writing `0x80` into the highest byte of the tag and the object's parent directory to the rest of the tag speeds up identifying the directory structure of a device.

4) blockSequence: The `blockSequence` tag is used to determine the chronological order in which chunks have been written to the NAND. Every time a new block is allocated for writing, the block sequence number is incremented and every chunk written to the block is tagged with this block sequence number. Thus, obsolete chunks can easily be detected and discarded. If two chunks with the same object numbers and chunk numbers can be found in different blocks, the chunk with the higher value in its `blockSequence` tag field is the most recent version. As can be seen in Listing 7, the value of the `blockSequence` tag can range from 4096 to 4 026 531 584. As depicted in Listing 2, a device's

```
1353 static int yaffs_FindNiceObjectBucket(yaffs_Device *dev)
1354 {
1355         int i;
1356         int l = 999;
1357         int lowest = 999999;
1358
1359
1360         /* Search for the shortest list or one that
1361          * isn't too long.
1362          */
1363
1364         for (i = 0; i < 10 && lowest > 4; i++) {
1365                 dev->bucketFinder++;
1366                 dev->bucketFinder %= YAFFS_NOBJECT_BUCKETS;
1367                 if (dev->objectBucket[dev->bucketFinder].count < lowest) {
1368                         lowest = dev->objectBucket[dev->bucketFinder].count;
1369                         l = dev->bucketFinder;
1370                 }
1371
1372         }
1373
1374         return l;
1375 }
1376
1377 static int yaffs_CreateNewObjectNumber(yaffs_Device *dev)
1378 {
1379         int bucket = yaffs_FindNiceObjectBucket(dev);
1380
1381         /* Now find an object value that has not already been taken
1382          * by scanning the list.
1383          */
1384
1385         int found = 0;
1386         struct ylist_head *i;
1387
1388         __u32 n = (__u32) bucket;
1389
1390         /* yaffs_CheckObjectHashSanity();  */
1391
1392         while (!found) {
1393                 found = 1;
1394                 n += YAFFS_NOBJECT_BUCKETS;
1395                 if (1 || dev->objectBucket[bucket].count > 0) {
1396                         ylist_for_each(i, &dev->objectBucket[bucket].list) {
1397                                 /* If there is already one in the list */
1398                                 if (i && ylist_entry(i, yaffs_Object,
1399                                                      hashLink)->objectId == n) {
1400                                         found = 0;
1401                                 }
1402                         }
1403                 }
1404         }
1405
1406         return n;
1407 }
```

Listing 4. Assignment of object numbers (Excerpt from yaffs_guts.c)

sequence number is incremented before a new block is allocated. Thus, the first block allocated by YAFFS2 on a NAND has the sequence number 4097.

5) nBytes: According to YAFFS2's specification [1], the two byte `nBytes` tag tells the number of data bytes of a chunk. However, during the analysis of YAFFS2, we discovered, that the `nBytes` tag can have several different meanings and sizes. This is the result of YAFFS2's use of its extended tags.

In the OOB area of a data chunk, the value of the `nBytes` tag defines how many bytes of data are contained within the chunk. In Figure 2, the OOB area of a 2048 bytes NAND page containing a data chunk is depicted. The `nBytes` value of `0x0800` shows, that this chunk is completely in use and holds 2048 bytes of data content. As the size of a chunk typically matches

```
/* Extra flags applied to chunkId */

#define EXTRA_HEADER_INFO_FLAG  0x80000000
#define EXTRA_SHRINK_FLAG       0x40000000
#define EXTRA_SHADOWS_FLAG      0x20000000
#define EXTRA_SPARE_FLAGS       0x10000000

#define ALL_EXTRA_FLAGS         0xF0000000

/* Also, the top 4 bits of the object Id are set to the object type. */
#define EXTRA_OBJECT_TYPE_SHIFT (28)
#define EXTRA_OBJECT_TYPE_MASK  ((0x0F) << EXTRA_OBJECT_TYPE_SHIFT)
[...]


void yaffs_PackTags2TagsPart(yaffs_PackedTags2TagsPart *ptt,
                const yaffs_ExtendedTags *t)
{
        ptt->chunkId = t->chunkId;
        ptt->sequenceNumber = t->sequenceNumber;
        ptt->byteCount = t->byteCount;
        ptt->objectId = t->objectId;

        if (t->chunkId == 0 && t->extraHeaderInfoAvailable) {
                /* Store the extra header info instead */
                /* We save the parent object in the chunkId */
                ptt->chunkId = EXTRA_HEADER_INFO_FLAG
                        | t->extraParentObjectId;
                if (t->extraIsShrinkHeader)
                        ptt->chunkId |= EXTRA_SHRINK_FLAG;
                if (t->extraShadows)
                        ptt->chunkId |= EXTRA_SHADOWS_FLAG;

                ptt->objectId &= ~EXTRA_OBJECT_TYPE_MASK;
                ptt->objectId |=
                    (t->extraObjectType << EXTRA_OBJECT_TYPE_SHIFT);

                if (t->extraObjectType == YAFFS_OBJECT_TYPE_HARDLINK)
                        ptt->byteCount = t->extraEquivalentObjectId;
                else if (t->extraObjectType == YAFFS_OBJECT_TYPE_FILE)
                        ptt->byteCount = t->extraFileLength;
                else
                        ptt->byteCount = 0;
        }

        yaffs_DumpPackedTags2TagsPart(ptt);
        yaffs_DumpTags2(t);
}
```

Listing 5. Extension of header chunks' tags (Excerpt from yaffs_packedtags2.c)

Object type	**Highest byte of the `objectID` tag value**
File	`0x10`
Soft link	`0x20`
Directory	`0x30`
Hard link	`0x40`
Special object (e.g. a pipe)	`0x50`

TABLE II
POSSIBLE VALUES OF THE `OBJECTID` TAG IN HEADER CHUNKS FOR DIFFERENT OBJECT TYPES

the size of NAND flash memories' pages, two bytes are sufficient for the `nBytes` tag of a data chunk.

In the OOB area of a header chunk, the meaning of the `nBytes` tag depends on the type of the object associated with the header chunk. As depicted in Listing 5 and Listing 6, in the OOB area of a file header chunk,

```
typedef struct {

        unsigned validMarker0;
        unsigned chunkUsed;     /*  Status of the chunk: used or unused */
        unsigned objectId;      /* If 0 then this is not part of an object (unused) */
        unsigned chunkId;       /* If 0 then this is a header, else a data chunk */
        unsigned byteCount;     /* Only valid for data chunks */

        /* The following stuff only has meaning when we read */
        yaffs_ECCResult eccResult;
        unsigned blockBad;

        /* YAFFS 1 stuff */
        unsigned chunkDeleted;  /* The chunk is marked deleted */
        unsigned serialNumber;  /* Yaffs1 2-bit serial number */

        /* YAFFS2 stuff */
        unsigned sequenceNumber;        /* The sequence number of this block */

        /* Extra info if this is an object header (YAFFS2 only) */

        unsigned extraHeaderInfoAvailable;      /* There is extra info available if this is not zero */
        unsigned extraParentObjectId;   /* The parent object */
        unsigned extraIsShrinkHeader;   /* Is it a shrink header? */
        unsigned extraShadows;          /* Does this shadow another object? */

        yaffs_ObjectType extraObjectType;       /* What object type? */

        unsigned extraFileLength;               /* Length if it is a file */
        unsigned extraEquivalentObjectId;       /* Equivalent object Id if it is a hard link */

        unsigned validMarker1;

} yaffs_ExtendedTags;
```

Listing 6. Extended tags structure (Excerpt from yaffs_guts.h)

```
#define YAFFS_LOWEST_SEQUENCE_NUMBER    0x00001000
#define YAFFS_HIGHEST_SEQUENCE_NUMBER   0xEFFFFF00
```

Listing 7. Possible values for the `blockSequence` tag (Excerpt from yaffs_guts.h)

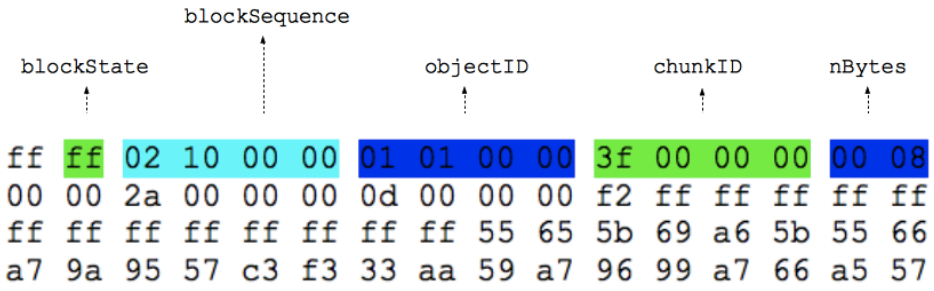


Fig. 2. OOB area of a data chunk

the `nBytes` tag stores the size of the file associated with the header chunk. As the size of a file can easily exceed the maximum value of 64 KB, two bytes are not sufficient to store a file's size. Therefore, as can be seen in Figure 3, four bytes are used for the `nBytes` tag of a file header chunk. The size of the file associated with the file header chunk depicted in Figure 3 amounts to 128 KB. Therefore, the value `0x00020000` is stored in

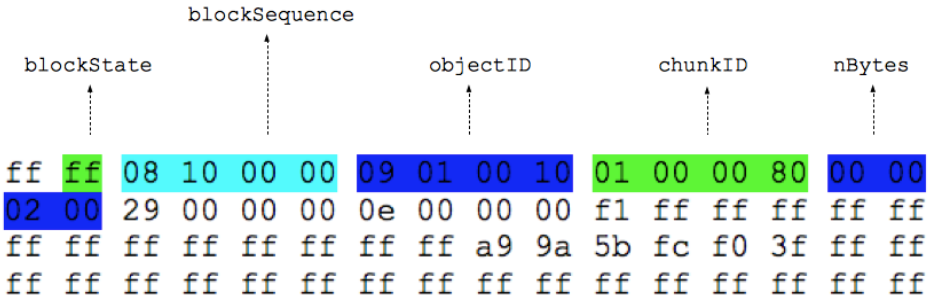


Fig. 3. OOB area of a file header chunk

the chunk's `nBytes` tag field.

In the header chunk of a hard link, the `nBytes` tag serves a completely different purpose. As a hard link just links another name to an existing file and does not have any data chunks, it does not need a `nByte` tag itself. Therefore the OOB area bytes that would normally contain the `nByte` flag, can be used for other information. As can be seen in line 87 and 88 of Listing 5, these bytes are used to store the object number of the object the hard link links to.

A soft link does also not feature any data chunks and consists only of one header chunk. In this header chunk, as can be seen in line 92 of Listing 5, the `nBytes` tag is not used and always contains the value zero. This is because a soft link uses the data area of the header chunk and not the chunk's OOB area for linking information. In figure 4, a chunk storing a soft link named *softlink2* is depicted. The soft link *softlink2* links to a file named *testFile1* that is stored in a directory named *aFolder*. As can be seen, the data area of a soft link's header chunk contains the absolute path of the object the soft link links to. The object type (`0x20`, marked blue) can be seen in Byte 0 of the chunk's data area as well as in the `objectID` tag in the chunk's OOB area. As can be seen in Bytes 4 to 7 (marked green) of the data area as well as in the `chunkID` tag in the OOB area, this soft link is stored in a directory with object number 1. As this object number is reserved for the root directory of a device, the soft link *softlink2* must be stored in the root directory.

6) tagsEcc and ecc: As NAND flash memory is prone to errors like bit flipping, error correction is necessary. Information regarding error correction is stored in the `tagsEcc` and `ecc` OOB area tags, with `tagsEcc` containing error correction information regarding a chunk's OOB area tags and `ecc` containing error correction information regarding a chunk's data contents.

7) Shrink header markers: Shrink header markers are YAFFS2's way to handle files with a hole. Such files occur when a file is truncated to a size smaller than its original size and a subsequent write operation on that file starts writing at an offset beyond the truncated size. As shrink header markers have an important influence on garbage collection, YAFFS2's way to handle files with holes are introduced in the following.

According to the POSIX standard, a hole inside a file should always read back as bytes of value zero [6]. As YAFFS2 does not rewrite already written chunks, YAFFS2 needs another way to create a hole inside a file and fill it with zeros. Additionally, data chunks that are located inside the hole after a truncation and a write operation have to be marked as obsolete. Depending on the size of a hole, YAFFS2 chooses one of two ways to handle files with a hole.

In case the hole is a hole smaller than four chunks (see lines 32 and 807 to 890 of `yaffs_yaffs2.c`) YAFFS2 actually writes zeros to the NAND to indicate the hole inside a file. This can be seen in Table III. The file *file.Hole* depicted in this NAND dump has been put through the following:

1) Writing of 15 000 'a' to the file, leading to a file size of 15 000 bytes
2) Truncation of the file to 1000 bytes
3) Writing of 3000 'b' at position 9191 of the file, leading to a hole of 8191 bytes and a new file size of 12 191 bytes

To represent the truncation of a file to a smaller size, YAFFS2 writes the data chunk that contains the new end of the file after the truncation to the NAND, followed by a file header chunk with the truncated size in its `nByte` OOB area tag. To represent the hole in the file, YAFFS2 then writes a respective number of chunks filled with zeros to the NAND, followed by chunks containing the file's content beyond the hole.

```
02 00 00 00 01 00 00 00 ff ff 73 6f 66 74 6c 69  |..........softli|
6e 6b 32 00 00 00 00 00 00 00 00 00 00 00 00 00  |nk2.............|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 ff ff ff a1 00 00  |................|
00 00 00 00 00 00 00 00 6c 73 6e 4d 6c 73 6e 4d  |........lsnMlsnM|
6c 73 6e 4d ff ff ff ff ff ff ff ff 61 46 6f 6c  |lsnM........aFol|
64 65 72 2f 74 65 73 74 46 69 6c 65 31 00 00 00  |der/testFile1...|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|
ff ff ff ff ff ff ff ff 00 00 00 00 ff ff ff ff  |................|
ff ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00  |................|
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|
                     ...
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|

ff ff 01 01 00 00 06 01 00 20 01 00 00 80 00 00
00 00 33 00 00 00 00 00 00 00 00 00 00 00 ff ff
ff ff ff ff ff ff ff ff fc c0 f3 f0 03 0f ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Fig. 4. Data area and OOB area of a soft link header chunk

In case the hole inside a file is bigger than four chunks, YAFFS2 does not write chunks completely filled with zeros to the NAND but makes use of its so-called shrink header markers to indicate a hole inside a file. As can be seen in Table IV, this saves a lot of space on the NAND. The file *file.Hole2* depicted in this NAND dump has been put through the following:

1) Writing of 15 000 'a' to the file, leading to a file size of 15 000 bytes
2) Truncation of the file to 1000 bytes
3) Writing of 3000 'b' at position 9192 of the file, leading to a hole of 8192 bytes (equivalent to four

Chunk no.	Content	chunkID	objectID	nBytes
...	...	...	...	...
8	file.Hole: Data (2048 'a')	1	258	2048
9	file.Hole: Data(2048 'a')	2	258	2048
10	file.Hole: Data (2048 'a')	3	258	2048
11	file.Hole: Data (2048 'a')	4	258	2048
12	file.Hole: Data (2048 'a')	5	258	2048
13	file.Hole: Data (2048 'a')	6	258	2048
14	file.Hole: Data (2048 'a')	7	258	2048
15	file.Hole: Data (664 'a', 1384 '0')	8	258	664
16	file.Hole: Data (1000 'a', 1048 '0')	1	258	1000
17	Header file.Hole	0x80000001	0x10000102	1000
18	Header file.Hole	0x80000001	0x10000102	1000
19	file.Hole: Data (1000 'a', 1048 '0')	1	258	2048
20	file.Hole: Data (2048 '0')	2	258	2048
21	file.Hole: Data (2048 '0')	3	258	2048
22	file.Hole: Data (2048 '0')	4	258	2048
23	file.Hole: Data (999 '0', 1049 'b')	5	258	2048
24	file.Hole: Data (1951 'b', 97 '0')	6	258	1951
25	file.Hole: Header	0x80000001	0x10000102	12191
...	...	...	...	...

TABLE III
REPRESENTATION OF A FILE WITH A HOLE SMALLER THAN FOUR CHUNKS ON A NAND

chunks of 2048 bytes) and a new file size of 12 192 bytes.

The header chunk marked with a shrink header marker, recognizable by its `chunkID` OOB area tag's highest byte's value of `0xC0`, marks the end of the hole inside the file and the data chunk written due to the truncation of the file (see chunk no. 16 in Table IV) the beginning of the hole. Therefore all of the file's chunks written before this chunk are obsolete. The size of the hole can be derived from the number of zeros written to the end respectively the beginning of the chunks enframing the hole and the gap between the `chunkID` tag values of the chunks enframing the hole.

B. Meta data

During a forensic analysis of a storage device, the device's files' meta data can provide valuable information. YAFFS2 uses an object's header chunk to store meta data such as time stamps, permissions and ownership information. This meta data of an object is not stored within the object header chunk's OOB area, but in the chunk's data area. In the following, we introduce the way YAFFS2 stores meta data.

1) Time stamps: Time stamps can provide relevant information about a file's history, that is, information about when a file has been accessed or modified. Typically, such information can be obtained from a file's `mtime`, `atime` and `ctime` time stamps. In a Linux environment, `mtime` stores the time a file's content has been modified, `atime` stores the time a file has been accessed and `ctime` the time a file's inode has been modified. Modifications of a file's inode include modifications of a file's meta data as well as modifications of a file's content.

YAFFS2 stores meta data such as time stamps inside a file header chunk's data area. However, regarding time stamps, YAFFS2 does not follow the default Unix or Linux way of keeping track of file modification and access times. During our analysis, we observed that YAFFS2 does store three time stamps inside an object's header chunk but these time stamps do not completely match classic Unix `mtime`, `atime` and `ctime` time stamps. Although YAFFS2 does store `ctime` and `mtime`, it does not store `atime` on a NAND. Instead of `atime` it stores a time stamp containing the time of the object's creation. On a simulated NAND as well as on a *HTC Magic* smartphone running Android OS 2.2, `mtime` could be found in Bytes 284 to 287 of an object's header chunk's data area, `ctime` could be found in Bytes 288 to 291 and the object's creation time in Bytes 280 to 283 of an object's header chunk's data area.

The reason for YAFFS2 waiver of writing `atime` to a NAND lies in wear considerations. Keeping track of an object's access times would require writing of a new object header chunk every time the object has

Chunk no.	Content	chunkID	objectID	nBytes
...	...	...	...	...
8	file.Hole2: Data (2048 'a')	1	258	2048
9	file.Hole2: Data (2048 'a')	2	258	2048
10	file.Hole2: Data (2048 'a')	3	258	2048
11	file.Hole2: Data (2048 'a')	4	258	2048
12	file.Hole2: Data (2048 'a')	5	258	2048
13	file.Hole2: Data (2048 'a')	6	258	2048
14	file.Hole2: Data(2048 'a')	7	258	2048
15	file.Hole2: Data (664 'a', 1384 '0')	8	258	664
16	file.Hole2: Data(1000 'a', 1048 '0')	1	258	1000
17	file.Hole2: Header	0x80000001	0x10000102	1000
18	file.Hole2: Header	0x80000001	0x10000102	1000
19	file.Hole2: Header	0xC0000001	0x10000102	1000
20	file.Hole2: Data (1000 '0', 1048 'b')	5	258	2048
21	file.Hole2: Data (1952 'b', 96 '0')	6	258	1952
22	file.Hole2: Header	0x80000001	0x10000102	12192
...	...	...	...	...

TABLE IV
REPRESENTATION OF A FILE WITH A HOLE OF FOUR CHUNKS ON A NAND

been accessed. This would lead to an enormous amount of object header chunks being written to the NAND which would lead to a drastic increase of block erasures and thus increased wear out of NAND flash memory. Because of that, by default, YAFFS2 does not write `atime` to a NAND [7].

2) Permissions and owner: In a Linux environment, every object, such as a file or a directory, has a user and a group that own the object. Read, write and execution permissions are assigned separately to the object's owner, the object's group and to all users that are neither the object's owner nor part of the group owning the object. Hence, for every object, the object's owner and group have to be stored along with the permissions granted to the owner, group and all other users.

On a simulated NAND as well as on a *HTC Magic* smartphone running Android OS 2.2, an object's permissions could be found in Bytes 268 to 271 of the object's header chunk's data area. The object's owner's UID (*User ID*) could be found in Bytes 272 to 275 of the object's header chunk's data area and the object's group's GID (*Group ID*) in Bytes 276 to 279.

C. Object modifications and deletions

As above-mentioned, YAFFS2 does not overwrite already written chunks. Thus, modification of an object's content or deletion of an object can not be performed by direct modification or deletion of the object's chunks on a NAND. Instead, existing chunks have to be marked as obsolete without overwriting these chunks on the NAND. In the following, we introduce YAFFS2's techniques to create, modify and delete objects.

1) Object creation: When creating an object, YAFFS2 first creates the object in RAM and then writes the object's chunks to the NAND. When writing the object's chunks to the NAND, YAFFS2 uses different writing patterns depending on the object's type. Hard links, soft links and directories do not feature any data chunks, but only consist of an object header chunk. When creating such an object on a NAND, YAFFS2 thus only writes the respective object header chunk. Additionally, a header chunk for the directory in which the newly created object is located in is written to the NAND to store the directory's changed meta data.

When creating a file, YAFFS2 not only has to write a file header chunk but also at least one data chunk to a NAND. To create the file on a NAND, YAFFS2 first writes a file header chunk with a `nByte` OOB area tag value of zero to indicate creation of an empty file. Subsequently, the file's data chunks are written to the NAND, followed by a file header chunk containing the actual file size. This way to create files on a NAND thus always leads to writing of a file header chunk that gets obsolete as soon as all the file's data chunks and the second file header chunk are written to the NAND. Finally, as with all other object types, a header chunk for the directory in which the newly created file is located in is written to the NAND to store the directory's changed meta data.

2) Object modifications: As above-mentioned, YAFFS2 keeps information about all objects stored in a RAM structure. This RAM structure does not only contain general information about an object, such as the object's type, name or object number, but also the object's *tnode tree*. Regarding modifications of objects, this tnode tree plays an important role. An object's tnode tree is used to map file positions to chunks on a NAND and thus is needed to determine which chunks need to be replaced in the course of an object's modification. When executing a modification of an object, YAFFS2 first modifies the object's meta information and tnode tree in RAM and then writes the respective modifications to the NAND.

YAFFS2 objects can be modified in several ways. Basically, there are two main categories of object modifications. Firstly, an object can be modified by modification of its meta data, such as permissions or time stamps. Secondly, objects can be modified by modification of their content. At that, the object's size can either stay the same, increase or decrease. These different cases of object modifications are handled differently by YAFFS2.

In case a modification only modifies an object's meta data, for example by use of `chmod`, only the object's header chunk is affected as no actual contents within the object's data chunks are modified. Thus, YAFFS2 only has to modify the object's header chunk. As YAFFS2 cannot overwrite the header chunk, YAFFS2 instead writes a new object header chunk containing the new meta data information. The new object header chunk contains the new meta data information and the same OOB area tags the old and now obsolete object header chunk features. As chunks and blocks are allocated for writing sequentially, YAFFS2, when scanning the device, discovers the new object header chunk before discovering the obsolete object header chunks. Thus, YAFFS2 recognizes which object header chunk has to be the current object header chunk and ignores the obsolete ones.

In case a modification changes an object's content, chunks containing the old content have to be marked as obsolete and new chunks containing the current content have to be written to the NAND. Additionally, a new object header chunk has to be written, as a modification of an object's content always leads to modification of an object's meta data. As only files contain data chunks, in the following, we only take modifications of files into consideration. In case modification of a file does not change the file's size, but only replaces parts of its contents, YAFFS2 needs to replace the affected data chunks. For this purpose, YAFFS2 determines which chunks have to be replaced by use of the file's tnode tree. The affected data chunks are then written to the NAND, containing the new content and the same OOB area tags the old and now obsolete data chunks feature. Additionally, a new file header chunk is written to store the file's modified meta data information and to indicate the modification. When scanning the NAND, YAFFS2 discovers the new file header chunk and new data chunks before discovering the obsolete file header and data chunks and thus considers the first discovered chunks current.

In case a modification changes a file's content in a way that changes the object's size, YAFFS2 additionally has to check whether the modification puts a hole into the file. Additionally, the `nByte` OOB area tag of the file header chunks has to be updated and, if necessary, also the `nByte` and `chunkID` OOB area tags of some data chunks. In case a modification puts a hole into a file, YAFFS2 needs to represent the hole on the NAND. YAFFS2's techniques to do so are described above in Section III-A7. In case a modification changes a file's size without resulting in a file with a hole, YAFFS2 does not have to write shrink header markers but has to consider whether the modification changes the file's data chunk count. Additionally, in case a modification that changes a file's size does not occur at the end of a file, the file positions of the data chunks located behind the point of modification have to be updated. Thus, the `chunkID` OOB area tags of these chunks have to be updated too.

In case a modification of a file occurs at the end of a file, either only the file's last data chunk or several data chunks at the end of the file are affected. If the modification is small enough only to change the last data chunk without changing the file's chunk count, YAFFS2 only has to rewrite the last data chunk. Thus, a new data chunk containing the new content is written to the NAND, featuring the same OOB area tags as the old last data chunk of the file with the only difference that the new data chunk's `nByte` OOB area tag has to contain the new number of bytes within the data chunk. Additionally, a new file header chunk featuring the new file size in its `nByte` OOB area tag has to be written. If a modification at the end of a file changes the file's data chunk count, YAFFS2 has to differentiate between modifications that increase the file's size and modifications that decrease the file's size. A modification that decreases a file's size and its data chunk count can either lead to a decrease that leads to a new end of

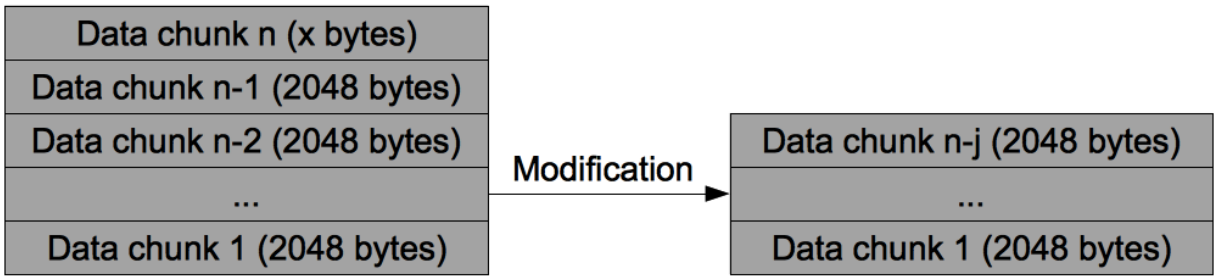


Fig. 5. Modification at the end of a file that decreases the file size to a multiple of a chunk's size

file that corresponds to the end of one of the file's data chunks or to a new end of file that is located within one of the file's data chunks. An end of file that corresponds to the end of one of the file's data chunks is given, if the new file size matches the multiple of a chunk's size. For example, as depicted in Figure 5, on a NAND with 2048 bytes per page, the new end of file after a file's modification corresponds to the end of one of the file's data chunks if the new file size is a multiple of 2048 bytes. Such a modification does not affect the content of the file's new last data chunk. Thus, as such a modification only truncates the file and does not affect the file's new last data chunk's contents, YAFFS2 only needs to write a new file header chunk containing the new file size in its `nByte` OOB tag. When scanning the device, YAFFS2 then considers all data chunks that lie behind the file's new last data chunk obsolete as, based on the file size given in the file's header chunk, they cannot be part of the file.

In case a modification at the end of a file leads to a decrease in file size that does not lead to the new end of file corresponding to the end of one of the file's data chunks, the contents of the file's new last data chunk is affected by the modification. Thus, in this case, YAFFS2 needs to rewrite the file's new last data chunk with new content and updated OOB area tags. Therefore, a new data chunk is written to the NAND, containing the new content and the same OOB area tags as the affected data chunk, except for the `nByte` tag that now features a smaller value. Additionally, a new file header chunk featuring the new file size in its `nByte` OOB area tag is written to the NAND.

Modifications at the end of a file that increase the file's size and the file's chunk count by definition append content to the end of the file. Thus, such a modification requires at least writing of one new chunk containing the new content and, if necessary, rewriting of some of the file's old chunks. Simply appending content to a file only requirers rewriting of the file's old last chunk if this chunk was not completely filled before the modification took place and thus parts of the content appended to the file are to be located inside this chunk. In this case, YAFFS2 writes a data chunk containing the file's old last chunk's content plus as much of the appended content to completely fill the chunk on the NAND. This chunk's OOB area tag values match the file's old last data chunk's tag values except for the `nByte` tag that now features a value representing a completely filled chunk. The remaining content to be appended to the file is written to additional new data chunks whose `chunkID` OOB area tags contain values placing them at the end of the file. If the modification does not only append content to the file but also replaces content located in several consecutive data chunks at the end of the file, this data chunks have to be rewritten too. Additionally, a new file header chunk containing the new file size in its `nByte` OOB area tag is written to the NAND.

Obviously, modifications of a file's content do not only occur at the end of a file, but also at other positions of a file. Such modifications require more of the file's data chunks to be rewritten as not only the data chunks whose content actually changes get modified but also the positions of all data chunks behind the point of the modification. This requires updating the `chunkID` OOB area tag of these chunks which again requires rewriting these chunks. If such a modification leads to hole in the file, the modification is handled by YAFFS2 as described in Section III-A7. In case a modification of a file's content at a position other than the end of the file does not change the file's size and causes no hole in the file, only the data chunks affected by the modification have to be rewritten. YAFFS2 thus writes new data chunks containing the new content and featuring the same OOB area tags as the replaced chunks to the NAND. Additionally, YAFFS2 writes a new file header chunk. If the modification also changes the file's

size, also all data chunks located behind the point of the modification are rewritten. This is because at least some of them feature new contents after the modification and all of them require updating of their `chunkID` OOB area tag.

3) Object deletions: As mentioned in Section III-A, YAFFS2 uses the pseudo objects `unlinked` and `deleted` to perform deletion of objects. These pseudo objects represent virtual directories that are not actually written to a NAND. When a link to an object is deleted, the link is virtually moved to the `deleted` directory. If an object is deleted and no links to the object exist anymore on the device, the object is virtually moved to the `unlinked` and `deleted` directory. The reason for that way of marking objects as deleted is YAFFS2's zero overwrite policy that prevents YAFFS2 from overwriting already written chunks without erasing the chunks' respective blocks beforehand. In the following, we introduce YAFFS2's way to mark objects as deleted. A practical evaluation of YAFFS2's behavior when performing delete operations is provided in the work of Zimmermann [8].

As above-mentioned, the `unlinked` and `deleted` directories do not actually exist on a NAND. Therefore, no object header chunks for these objects are written to the NAND. Hence, to move an object to the `deleted` or `unlinked` directory, a special `unlinked` or `deleted` object header chunk for the deleted or unlinked object is written to the NAND. After writing of a `deleted` or an `unlinked` header chunk, a directory header chunk for the directory containing the deleted object is also written to the NAND to update the directory's time stamps. Although the `unlinked` and `deleted` header chunks resemble regular object header chunks in many ways, there are some differences. In Table V the way YAFFS2 uses a `deleted` header chunk's OOB area tags is depicted. All OOB area tags not listed in Table V are used as they are in the deleted object's header chunk.

As shown in Section III-A and depicted in Listing 3, object number 4 is reserved for the `deleted` directory. Thus, as can be seen in Table V, a `deleted` header chunk resembles the object header chunk of an object located inside the `deleted` directory. However, some differences are discernible. The `deleted` header chunk's `chunkID` OOB area tag is also marked with a shrink header marker. Although shrink header markers are typically used to mark files with a hole in them, they are also used to indicate deletion of an object. Thus, deletion of an object can also be seen as a resizing of the object to a size of zero bytes. The `nByte` OOB area tag of a `deleted` header chunk does either contain the value zero or an object number. The `nByte` OOB area tag value is always set to zero, if an object gets deleted and no links to the object are present on the device anymore. If a hard link gets deleted and the object it links to is still present on the device, the hard link's `deleted` header chunk's `nByte` OOB area tag features the object number of the object the hard link linked to.

Usually, an object header chunk features the object's name in the object header chunk's data area. Although a `deleted` header chunk features the same value in its `objectID` OOB area tag as the deleted object, the name stored in the `deleted` header chunk is always "deleted".

When all links to an object have been deleted, the object is not only virtually moved to the `deleted` directory but also to the `unlinked` directory to indicate the object's complete deletion. For example, if a file "*xFile*" and a hard link to this file exist on a device, deletion of the hard link leads to a `deleted` header chunk for the hard link being written to the NAND. If subsequently the file "*xFile*" gets deleted both an `unlinked` and a `deleted` header chunk for the file are written to the NAND. Further analysis of YAFFS2's behavior in handling deletion of hard links and the objects they link to is provided below.

As can be seen in Table VI, an `unlinked` header chunk's OOB area tag values strongly resemble a `deleted` header chunk's OOB area tag values. The main difference between `unlinked` and `deleted` header chunks lies in their different `chunkID` OOB area tag values. Additionally, an `unlinked` header chunk features the name "unlinked" instead of "deleted" in its data area. As shown in Section III-A and depicted in Listing 3, the virtual `unlinked` directory has the special object number 3. Thus, writing an `unlinked` header chunk for an object and thus virtually moving the object to the virtual `unlinked` directory requires the object number "3" being written to the `unlinked` header chunk's `chunkID` OOB area tag as the object number of the deleted object's parent directory. As an `unlinked` header chunk is alway written to NAND flash memory together with a `deleted` header chunk, an `unlinked` header chunk does not need a shrink header marker.

Technically, a file name represents a link to a file object. If no link to this object other than the file name exists, deletion of the object is performed by writing `unlinked` and `deleted` header chunks for the object.

OOB area tag	content
objectID	value of the deleted object's objectID OOB area tag
chunkID	0xc0000004
nBytes	"0" or the deleted object's object number

TABLE V
USAGE OF OOB AREA TAGS IN DELETED HEADER CHUNKS

OOB area tag	content
objectID	value of the unlinked object's objectID OOB area tag
chunkID	0x80000003
nBytes	0

TABLE VI
USAGE OF OOB AREA TAGS IN UNLINKED HEADER CHUNKS

But in case hard links to the file object exist on a device, deletion of the file does only delete the file name and not the underlying object as links to the object still exist on the device. The way YAFFS2 handles this problem is depicted in Table VII.

In Table VII, a dump of a NAND on which the following actions have been performed is depicted:

1) Creation of file *"testFile"* (object number 257)
2) Creation of hard link *"hardLink1"* to *"testFile"* (object number 260)
3) Creation of hard link *"hardLink2"* to *"testFile"* (object number 261)
4) Deletion of *"hardLink2"*
5) Deletion of *"testFile"*
6) Deletion of *"hardLink1"*

As can be seen in Table VII, deletion of *"hardLink2"* is performed as described above by writing a deleted header chunk for *"hardLink2"* to the NAND (see chunk no. 15 in Table VII). However, deletion of *"testFile"* does not result in writing of an unlinked or deleted header chunk for the file object but to writing of a new file header chunk for the file object with object number 257 and deletion of *"hardLink1"*. The reason for that is, that both the file name *"testFile"* and the hard link *"hardLink1"* link to the same file object and thus deletion of *"testFile"* can not result in actual deletion of the file object. Instead, only the link to the file object represented by the file name *"testFile"* is deleted. Thus, YAFFS2 writes a new file header chunk for the file object with object number 257 to the NAND, thereby renaming the file object to *"hardLink1"* (see chunk no. 17 in Table VII). As the name *"hardLink1"* is now the only remaining link to the file object, the hard link object with object number 260 has to be deleted by writing a deleted header chunk for this object (see chunk no. 18 in Table VII). Only after the file name *"hardLink1"* is deleted, the file object with object number 257 is finally actually deleted. As no links to the object exist anymore, it is deleted by writing an unlinked and a deleted header chunk for the object to the NAND (see chunk no. 20 and 21 in Table VII).

IV. GARBAGE COLLECTION

To comply to NAND flash memory's demands, YAFFS2 never overwrites an already written chunk without deleting the chunk's block beforehand. Hence, chunks whose content becomes obsolete through modification or deletion of their respective objects remain stored on the NAND until the blocks containing these chunks are deleted. Obsolete chunks need to be deleted at some point to free up space on the device. While it is not a problem to delete a block that is completely filled with obsolete chunks, simply deleting a block filled with a mixture of obsolete and current chunks would lead to data loss. Thus, YAFFS2 needs a way to ensure that all current chunks within a block that is to be deleted are copied to another block before performing the deletion. Saving these current chunks is task of YAFFS2's garbage collector. From a forensic perspective, as obsolete chunks contain potential evidence, it is important to understand the way, YAFFS2's garbage collector performs its task. Therefore, in the following, we provide an in-depth analysis of the YAFFS2 garbage collector.

YAFFS2 distinguishes between two different modes of garbage collection, *aggressive* garbage collection and *passive* garbage collection [4]. Which of these modes is used is decided in the first of three steps of

Chunk no.	Content	chunkID	objectID	nBytes
0	testFile: Header	0x80000001	0x10000101	0
1	testFile: Data	1	257	2048
2	testFile: Header	0x80000001	0x10000101	2048
...	...	...	...	...
8	hardLink1: Header	0x80000001	0x40000104	257
...	...	...	...	...
10	hardLink2: Header	0x80000001	0x40000105	257
...	...	...	...	...
15	deleted: Header	0xC0000004	0x40000105	257
...	...	...	...	...
17	hardLink1: Header	0x80000001	0x10000101	2048
18	deleted: Header	0xC0000004	0x40000104	257
...	...	...	...	...
20	unlinked: Header	0x80000003	0x10000101	0
21	deleted: Header	0xC0000004	0x10000101	0
...	...	...	...	...

TABLE VII
DELETION OF A FILE WITH HARD LINKS ON A YAFFS2 NAND

YAFFS2's garbage collection. YAFFS2's garbage collector's functionality is defined in files `yaffs_guts.c` and `yaffs_yaffs2.c` and features four main functions representing three major steps of garbage collection. These functions are:

- `yaffs_CheckGarbageCollection` (lines 2544 to 2632 of `yaffs_guts.c`, Listing 8, 9)
- `yaffs_FindBlockForGarbageCollection` (lines 2379 to 2533 of `yaffs_guts.c`)
- `yaffs2_FindRefreshBlock` (lines 142 to 192 of `yaffs_yaffs2.c`),
- `yaffs_GarbageCollectBlock` (lines 2102 to 2372 of `yaffs_guts.c`)

Every garbage collection starts with a call of `yaffs_CheckGarbageCollection` as a first step in which necessity of garbage collection is determined. Execution of `yaffs_CheckGarbageCollection` results in either abortion of the current attempt of garbage collection or in a call of `yaffs_FindBlockForGarbageCollection` or `yaffs2_FindRefreshBlock` to find a block for garbage collection. After a suitable block for garbage collection has been found, actual garbage collection of this block is performed as third step of garbage collection by use of `yaffs_GarbageCollectBlock`.

Function `yaffs_CheckGarbageCollection` is called to check whether garbage collection is necessary every time one of the following functions is called:

- `yaffs_BackgroundGarbageCollect` (lines 2639 to 2647 of `yaffs_guts.c`)
- `yaffs_WriteChunkDataToObject` (lines 2936 to 2994 of `yaffs_guts.c`)
- `yaffs_updateObjectHeader` (lines 2999 to 2171 of `yaffs_guts.c`)
- `yaffs_ResizeFile` (lines 3792 to 3847 of `yaffs_guts.c`)

At least one of these functions is called every time a file system operation requires writing of data to the NAND. Thus, YAFFS2 checks necessity of garbage collection every time data has to be written to the NAND. That does not only include creation of objects, but also deletion or modification of existing objects. Additionally, YAFFS2 checks for necessity of garbage collection at fixed time intervals by use of function `yaffs_BackgroundGarbageCollect`. On our test system, we observed checks for necessity of garbage collection by `yaffs_BackgroundGarbageCollect` every two seconds. In case of a NAND that was never mounted before, the first check for necessity of garbage collection by `yaffs_BackgroundGarbageCollect` was performed directly after mounting of the NAND. In case the NAND had been mounted before, the first check for necessity of garbage collection by `yaffs_BackgroundGarbageCollect` was performed after the first write operation to the NAND. Whether garbage collection of a block is actually initiated after a check for necessity for garbage collection and, if it is, in which mode it is performed,

depends on several factors. First of all, YAFFS2 checks, whether garbage collection is permanently or temporarily deactivated for a device (see line 2553 to 2560 of Listing 8). Garbage collection is temporarily deactivated during garbage collection of a block to prevent recursive garbage collection. As can be seen in Listing 8, step two of garbage collection is always performed if aggressive garbage collection proves necessary. This is the case, if not enough free blocks are available to store a checkpoint. As defined in function `yaffs2_CalcCheckpointBlocksRequired` (lines 213 to 247 of `yaffs_yaffs2.c`) and line 2571 of `yaffs_guts.c`, at least n free blocks must be available for checkpoint data in order not to trigger aggressive garbage collection, with n having the following value:

n = number of reserved blocks
+ number of complete blocks actually necessary to store current checkpoint data
− number of blocks currently used for checkpoint data
+ 4

If at least n free blocks are available to store current checkpoint data and for that reason no aggressive garbage collection has to be performed, YAFFS2 checks, whether passive garbage collection is necessary. Execution of passive garbage collection depends on two factors. These are:

1) the way the garbage collection check has been invoked
2) the number of free chunks within free blocks in relation to the total number of free chunks on the device

Regarding passive garbage collection, YAFFS2 distinguishes between garbage collection checks that have been caused by background threads and garbage collection checks that have been caused by foreground threads. In the following, garbage collection that is caused by a background thread is also referred to as *background garbage collection*. Typically, YAFFS2's periodical check for necessity of garbage collection is the main trigger for background garbage collection. As can be seen in Listing 8, once aggressive garbage collection proved unnecessary, passive garbage collection is performed unless the `if`-statement in line 2578 returns `true` or the `if`-statement in line 2583 returns `false`. If a check for necessity of garbage collection is caused by a background thread, variable `background` is set to value 1. This always leads to the `if`-statement in line 2578 returning `false` and the `if`-statement in line 2583 returning `true` and thus, given that aggressive garbage collection is not necessary, to execution of step two of passive garbage collection. If a check for necessity of garbage collection is caused by a foreground thread, variable `background` is set to value 0. This means, that the `if`-statement in line 2578 can only return `false` if the number of free chunks within free blocks does not exceed one quarter of the total number of free chunks on the device. This automatically leads to the `if`-statement in line 2583 returning `true`, as, if the number of free chunks within free blocks does not exceed one quarter of the total number of free chunks on the device, it cannot exceed half the total number of free chunks on the device. Hence, if a check for necessity of garbage collection is caused by a foreground thread and aggressive garbage collection is not necessary, execution of step two of passive garbage collection depends only on the ratio of free chunks within free blocks to the total number of free chunks on the device.

Thus, a check for necessity of garbage collection leads to further steps of garbage collection if garbage collection is not deactivated and at least one of the following conditions is met:

- shortage of free blocks would prevent storing of checkpoint data
- garbage collection is initiated from a background thread
- garbage collection is initiated from a foreground thread and the number of free chunks within free blocks does not exceed one quarter of the total number of free chunks on the device

As can be seen in Listing 8, YAFFS2 checks for necessity of garbage collection up to two times. However, the second check is only performed if the first check leads to garbage collection and garbage collection does not free up enough space. If, after a first garbage collection, the number of free blocks is smaller than the number of blocks reserved for checkpoint data and a block for garbage collection is still available, a second check is performed. This second check always leads to aggressive garbage collection, as the number of blocks reserved for checkpoint data is always smaller than the value of variable `minErased` (see line 2571 of Listing 8), because function `yaffs2_CalcCheckPointBlocksRequired` of `yaffs_yaffs2.c` never returns a value smaller than 0. A second check can only occur if the number of free blocks is smaller than the number of blocks reserved for checkpoint data. Aggressive garbage collection is

```
2544 static int yaffs_CheckGarbageCollection(yaffs_Device *dev, int background)
2545 {
2546         int aggressive = 0;
2547         int gcOk = YAFFS_OK;
2548         int maxTries = 0;
2549         int minErased;
2550         int erasedChunks;
2551         int checkpointBlockAdjust;
2552
2553         if(dev->param.gcControl &&
2554                 (dev->param.gcControl(dev) & 1) == 0)
2555                 return YAFFS_OK;
2556
2557         if (dev->gcDisable) {
2558                 /* Bail out so we don't get recursive gc */
2559                 return YAFFS_OK;
2560         }
2561
2562         /* This loop should pass the first time.
2563          * We'll only see looping here if the collection does not increase space.
2564          */
2565
2566         do {
2567                 maxTries++;
2568
2569                 checkpointBlockAdjust = yaffs2_CalcCheckpointBlocksRequired(dev);
2570
2571                 minErased  = dev->param.nReservedBlocks + checkpointBlockAdjust + 1;
2572                 erasedChunks = dev->nErasedBlocks * dev->param.nChunksPerBlock;
2573
2574                 /* If we need a block soon then do aggressive gc.*/
2575                 if (dev->nErasedBlocks < minErased)
2576                         aggressive = 1;
2577                 else {
2578                         if(!background && erasedChunks > (dev->nFreeChunks / 4))
2579                                 break;
2580
2581                         if(dev->gcSkip > 20)
2582                                 dev->gcSkip = 20;
2583                         if(erasedChunks < dev->nFreeChunks/2 ||
2584                                 dev->gcSkip < 1 ||
2585                                 background)
2586                                 aggressive = 0;
2587                         else {
2588                                 dev->gcSkip--;
2589                                 break;
2590                         }
2591                 }
2592
2593                 dev->gcSkip = 5;
2594
2595                 /* If we don't already have a block being gc'd then see if we should start another */
2596
2597                 if (dev->gcBlock < 1 && !aggressive) {
2598                         dev->gcBlock = yaffs2_FindRefreshBlock(dev);
2599                         dev->gcChunk = 0;
2600                         dev->nCleanups=0;
2601                 }
2602                 if (dev->gcBlock < 1) {
2603                         dev->gcBlock = yaffs_FindBlockForGarbageCollection(dev, aggressive, background)
                                ;
2604                         dev->gcChunk = 0;
2605                         dev->nCleanups=0;
2606                 }
```

Listing 8. Function `yaffs_CheckGarbageCollection` Part 1 (Excerpt from yaffs_guts.c)

```
2607
2608                if (dev->gcBlock > 0) {
2609                        dev->allGCs++;
2610                        if (!aggressive)
2611                                dev->passiveGCs++;
2612
2613                        T(YAFFS_TRACE_GC,
2614                          (TSTR
2615                           ("yaffs: GC erasedBlocks %d aggressive %d" TENDSTR),
2616                           dev->nErasedBlocks, aggressive));
2617
2618                        gcOk = yaffs_GarbageCollectBlock(dev, dev->gcBlock, aggressive);
2619                }
2620
2621                if (dev->nErasedBlocks < (dev->param.nReservedBlocks) && dev->gcBlock > 0) {
2622                        T(YAFFS_TRACE_GC,
2623                          (TSTR
2624                           ("yaffs: GC !!!no reclaim!!! erasedBlocks %d after try %d block %d"
2625                            TENDSTR), dev->nErasedBlocks, maxTries, dev->gcBlock));
2626                }
2627        } while ((dev->nErasedBlocks < dev->param.nReservedBlocks) &&
2628                 (dev->gcBlock > 0) &&
2629                 (maxTries < 2));
2630
2631        return aggressive ? gcOk : YAFFS_OK;
2632 }
2633
2634 }
```

Listing 9. Function `yaffs_CheckGarbageCollection` Part 2 (Excerpt from yaffs_guts.c)

performed if the number of free blocks is smaller than the value of variable `minErased` which is always bigger than the number of blocks reserved for checkpoint data. Therefore, the `if`-statement in line 2575 of Listing 8 always returns `true` during a second check, thus causing aggressive garbage collection.

After checking necessity of garbage collection, selection of a block to actually garbage collect is performed as second step of every garbage collection. This step is skipped if a block for garbage collection has already been selected in an earlier garbage collection cycle and has not yet been completely garbage collected. If no block for garbage collection has been selected, either function `yaffs_FindBlockForGarbageCollection` or function `yaffs2_FindRefreshBlock` is called to select a block for garbage collection. If passive garbage collection is performed, first function `yaffs2_FindRefreshBlock` of `yaffs_yaffs2.c` is called. This function's purpose is to enable *block refreshing*, a wear leveling technique. Function `yaffs2_FindRefreshBlock` returns the oldest block in state `FULL` with the oldest block being the block with the lowest sequence number and thus the block that has not been written to for longer than any other block. However, `yaffs2_FindRefreshBlock` only returns this block if a fixed number of blocks have been selected for garbage collection beforehand by function `yaffs_FindBlockForGarbage Collection`. By default, `yaffs2_FindRefreshBlock` only returns the oldest block every 500 executions of `yaffs_FindBlockForGarbageCollection` that actually lead to a block being selected for garbage collection. Otherwise, `yaffs2_FindRefreshBlock` returns 0 and therefore does not select a block for garbage collection. If `yaffs2_FindRefreshBlock` returns the value 0 or aggressive garbage collection has been chosen, function `yaffs_Find BlockForGarbageCollection` is used to find a block for garbage collection.

In function `yaffs_FindBlockForGarbage Collection`, most of the main differences between aggressive and passive garbage collection are defined. When trying to select a block for garbage collection, passive and aggressive garbage collection differ in three points, which are:

- the consideration of prioritized blocks.
- the intensity with which a block to garbage collect is searched for
- the number of obsolete chunks inside a block that are necessary to make the block a candidate for

garbage collection

YAFFS2 marks a block as prioritized for garbage collection if the block shows abnormal behavior, such as errors on read or write operations or a failed ECC check. As such errors can indicate a forthcoming failure of a block, its prioritization for garbage collection can prevent data loss through copying the block's contents to the block currently selected for allocation. However, when choosing a block to garbage collect, YAFFS2 only takes prioritizations into consideration when garbage collection is performed passively. When trying to find a block for passive garbage collection, YAFFS2, as can be seen in lines 2392 to 2408 of Listing 10, always selects the first prioritized block of a device for garbage collection, unless this block is not in state `FULL` or is disqualified for garbage collection. A block is disqualified for garbage collection if a file header chunk with a shrink header marker can be found within the block and the block's sequence number is higher than another block's sequence number and this block is in state `FULL` and features at least on obsolete chunk. Hence, a block featuring a header chunk marked with a shrink header marker is disqualified for garbage collection until it becomes the oldest of all blocks in state `FULL` that contain at least one obsolete chunk. If all prioritized blocks are disqualified for garbage collection, YAFFS2 tries to select the oldest block with obsolete chunks for passive garbage collection.

In case garbage collection is performed aggressively or passive garbage collection failed to select a block during its check for prioritized blocks in lines 2392 to 2423 of Listing 10, YAFFS2 starts a more extensive search for a block to garbage collect. The intensity of this search depends solely on whether garbage collection is performed aggressively or passively. However, how many obsolete chunks a block has to feature before becoming a valid candidate for garbage collection, does not only depend on whether garbage collection is performed aggressively or passively but, in case garbage collection is performed passively, also on whether garbage collection is triggered by a background thread or not. As garbage collection is performed aggressively only if not enough free blocks are available to store checkpoint data, aggressive garbage collection's goal is to free up space as quickly as possible. Therefore, when garbage collection is performed aggressively, YAFFS2 searches much harder for a block to garbage collect than is does when garbage collection is performed passively. As can be see in line 2436 and lines 2460 to 2481 of Listing 11, aggressive garbage collection, if necessary, checks all of a device's blocks for their suitability to be garbage collected. Passive garbage collection does not need to free up space as quickly as aggressive garbage collection and thus does not search for a block to garbage collect as intensely as aggressive garbage collection. Passive garbage collection checks at least one-sixteenth of all a device's blocks plus one block but maximal 100 blocks. Both aggressive and passive garbage collection check a device's blocks sequentially starting from the block the last search for a block to garbage collect has stopped at. As can be seen in lines 2474 to 2480 of Listing 11, both aggressive and passive garbage collection first select the best block for garbage collection from those blocks getting checked, which in case of aggressive garbage collection are all blocks. A block is suitable for garbage collection if it is in state `FULL`, has obsolete chunks and is not disqualified for garbage collection. From all blocks checked that meet these requirements the block featuring the most obsolete chunks is selected as the best candidate for being garbage collected.

If this best candidate is finally actually selected to be garbage collected, again, depends on the mode garbage collection is performed in. As can be seen in line 2483 of Listing 11, before a block is selected to be garbage collected, one last check is performed. Because of this check, the best candidate for garbage is only selected to be garbage collected if the number of its chunks containing valid data does not exceed a a certain *threshold*. The actual value of this threshold depends on the mode garbage collection is performed in. In case garbage collection is performed aggressively, the threshold value equals the number of chunks per block so that the best candidate is always selected to be garbage collected, even if it features only one single obsolete chunk. As passive garbage collection does not need to free up space as quickly as aggressive garbage collection, it has no need to garbage collect blocks that feature only a very small amount of obsolete chunks. Thus, the threshold is set to a smaller value when garbage collection is performed passively. As can be seen in lines 2438 to 2458 of Listing 11, the value used as threshold also depends on whether passive garbage collection is a background garbage collection or not. In case passive garbage collection is a background garbage collection, the threshold is set to at least double the number of skipped garbage collections increased by 2 and maximal one half of the number of chunks per block. Thus, background garbage collection accepts a higher number of valid chunks inside a block to be garbage collected after every

skipped garbage collection but never garbage collects a block that has more than half of its chunks in use. If the value chosen for the threshold is smaller than the value of YAFFS_GC_PASSIVE_THRESHOLD, which by default is four, YAFFS_GC_PASSIVE_THRESHOLD is used as threshold. A garbage collection is skipped, if none of the blocks checked meets all requirements to be garbage collected. As seen above, this can only happen in case garbage collection is performed passively. In case passive garbage collection is not a background garbage collection the threshold is set to at least the value of YAFFS_GC_PASSIVE_THRESHOLD and maximal one-eighth of the number of chunks per block. The combination of passive garbage collection's limited intensity of search for a block to garbage collect and the threshold requirement is the reason that passive garbage collection does not always find a block to garbage collect and thus is skipped. As passive garbage collection only checks a subset of all blocks for their suitability to be garbage collected it is likely that no suitable block is found. Additionally, if a suitable block is found there is still no guarantee that this block is actually selected to be garbage collected because of the threshold requirement. To mitigate this problem, YAFFS2 keeps track of the number of consecutively skipped garbage collections. If passive garbage collection has been consecutively skipped often enough, YAFFS2 tries to select the device's oldest block in state FULL that features at least one obsolete chunk to be garbage collected. As can be seen in lines 2492 to 2503 of listing 12, for this to happen, passive garbage collection must have been skipped ten consecutive times in case of background garbage collection and twenty consecutive times in case garbage collection was triggered by a foreground thread. In the following, this way to select a block for garbage collection is also referred to as *oldest dirty garbage collection*.

As seen above, YAFFS2 puts a lot of effort in selecting a block for garbage collection and knows a variety of reasons why not to garbage collect a specific block. Nonetheless, a basic goal in selecting a block to garbage collect is recognizable. This is the goal to select the dirtiest and oldest block possible for garbage collection.

After checking for necessity of garbage collection and selecting a block to garbage collect, the third and last step of every garbage collection cycle consists of actually copying valid chunks from the selected block to the block currently allocated for write operations. This task is performed by function yaffs_GarbageCollectBlock (lines 2102 to 2372 of yaffs_guts.c). This function shows the last important difference between aggressive and passive garbage collection. While aggressive garbage collection collects the whole block at one go, passive garbage collection only collects five valid chunks per garbage collection cycle. Because of that, passive garbage collection can need several garbage collection cycles to collect a block. Once a block has been completely collected, its state is set to DIRTY which leads to its immediate deletion and transition into state EMPTY. Additionally, if the block selected for garbage collection is a block containing checkpoint data, the block's state is also set to DIRTY immediately and no chunks are copied off.

V. Wear leveling

One of the issues that a NAND flash filesystem has to deal with, is flash memory's limited endurance. Flash memory's erase blocks can only endure a limited number of erase and rewrite cycles, typically ranging from 10^4 to 10^6 cycles [9]. In order to prevent single blocks to wear out faster than the device's other blocks, write and erase operations have to be evenly distributed among all of a flash memory's blocks. Techniques to achieve such a distribution of write and erase cycles are referred to as wear leveling and are an important feature of flash memory file systems.

Basically, a flash memory file system has two options to perform wear leveling. These can be described as explicit and implicit wear leveling techniques. A flash file system performing explicit wear leveling features special functionality exclusively dedicated to perform wear leveling, whereas a flash file system performing implicit wear leveling does not feature any special functionality regarding wear leveling. Implicit wear leveling relies on the basic design of the flash file system to evenly distribute erase and write load among the flash memory's blocks.

YAFFS2 performs wear leveling mainly implicitly but also features one explicit wear leveling technique, block refreshing. YAFFS2's design implicitly supports wear leveling in several ways. First of all, YAFFS2 does not use any central structures, such as a file allocation table. Therefore, YAFFS2 has no need to write such data to fixed addresses on the flash memory, hence preventing that blocks at these addresses wear out much faster than other blocks of the flash memory. Additionally to not writing specific blocks excessively often, YAFFS2 distributes its write operations evenly by means of its block allocation policy. As already depicted in Listing 2, as long as free blocks exist on a flash memory,

```
2379 static unsigned yaffs_FindBlockForGarbageCollection(yaffs_Device *dev,
2380                                                     int aggressive,
2381                                                     int background)
2382 {
2383         int i;
2384         int iterations;
2385         unsigned selected = 0;
2386         int prioritised = 0;
2387         int prioritisedExists = 0;
2388         yaffs_BlockInfo *bi;
2389         int threshold;
2390
2391         /* First let's see if we need to grab a prioritised block */
2392         if (dev->hasPendingPrioritisedGCs && !aggressive) {
2393                 dev->gcDirtiest = 0;
2394                 bi = dev->blockInfo;
2395                 for (i = dev->internalStartBlock;
2396                         i <= dev->internalEndBlock && !selected;
2397                         i++) {
2398
2399                         if (bi->gcPrioritise) {
2400                                 prioritisedExists = 1;
2401                                 if (bi->blockState == YAFFS_BLOCK_STATE_FULL &&
2402                                    yaffs2_BlockNotDisqualifiedFromGC(dev, bi)) {
2403                                         selected = i;
2404                                         prioritised = 1;
2405                                 }
2406                         }
2407                         bi++;
2408                 }
2409
2410                 /*
2411                  * If there is a prioritised block and none was selected then
2412                  * this happened because there is at least one old dirty block gumming
2413                  * up the works. Let's gc the oldest dirty block.
2414                  */
2415
2416                 if(prioritisedExists &&
2417                         !selected &&
2418                         dev->oldestDirtyBlock > 0)
2419                         selected = dev->oldestDirtyBlock;
2420
2421                 if (!prioritisedExists) /* None found, so we can clear this */
2422                         dev->hasPendingPrioritisedGCs = 0;
2423         }
```

Listing 10. Function `yaffs_FindBlockForGarbageCollection` Part 1 (Excerpt fromyaffs_guts.c)

YAFFS2, being a truly log-structured file system, tries to allocate them in sequential order. Thus, YAFFS2 achieves a certain level of wear leveling solely implicitly. Additionally, YAFFS2 can use its block refreshing technique to distribute block usage evenly among all blocks of a NAND. Block refreshing, in short, tries to spread wear by moving the oldest full block's content to other blocks and erasing the oldest block. In doing so, block refreshing ensures that every block is erased at some point, even if no obsolete chunks can be found on that block and thus the block is not a candidate for regular garbage collection. Although block refreshing can be disabled at compile time, it is enabled by default. As already mentioned in Section IV, block refreshing is a task performed by YAFFS2's garbage collector. As can be seen in line 2597 of Listing 8, block refreshing can only occur, if a check for necessity of garbage collection proves passive garbage collection necessary and no block has already been selected to be garbage collected. As can be seen in Listing 13, `yaffs2_FindRefreshBlock` only selects the oldest block for refreshing, if variable `refreshSkip` has value zero. By default, this variable is set to value zero on mount of a YAFFS2 device [10] and to a value of 500 after the first block refreshing. The value of `refreshSkip` is only decremented in function `yaffs_FindBlockForGarbageCollection` in case a block for garbage collection is selected (see Listing 10). Thus, block refreshing can only occur as

```
2424
2425        /* If we're doing aggressive GC then we are happy to take a less-dirty block, and
2426         * search harder.
2427         * else (we're doing a leasurely gc), then we only bother to do this if the
2428         * block has only a few pages in use.
2429         */
2430
2431        if (!selected){
2432                int pagesUsed;
2433                int nBlocks = dev->internalEndBlock - dev->internalStartBlock + 1;
2434                if (aggressive){
2435                        threshold = dev->param.nChunksPerBlock;
2436                        iterations = nBlocks;
2437                } else {
2438                        int maxThreshold;
2439
2440                        if(background)
2441                                maxThreshold = dev->param.nChunksPerBlock/2;
2442                        else
2443                                maxThreshold = dev->param.nChunksPerBlock/8;
2444
2445                        if(maxThreshold <  YAFFS_GC_PASSIVE_THRESHOLD)
2446                                maxThreshold = YAFFS_GC_PASSIVE_THRESHOLD;
2447
2448                        threshold = background ?
2449                                 (dev->gcNotDone + 2) * 2 : 0;
2450                        if(threshold <YAFFS_GC_PASSIVE_THRESHOLD)
2451                                threshold = YAFFS_GC_PASSIVE_THRESHOLD;
2452                        if(threshold > maxThreshold)
2453                                threshold = maxThreshold;
2454
2455                        iterations = nBlocks / 16 + 1;
2456                        if (iterations > 100)
2457                                iterations = 100;
2458                }
2459
2460                for (i = 0;
2461                        i < iterations &&
2462                        (dev->gcDirtiest < 1 ||
2463                                dev->gcPagesInUse > YAFFS_GC_GOOD_ENOUGH);
2464                        i++) {
2465                        dev->gcBlockFinder++;
2466                        if (dev->gcBlockFinder < dev->internalStartBlock ||
2467                                dev->gcBlockFinder > dev->internalEndBlock)
2468                                dev->gcBlockFinder = dev->internalStartBlock;
2469
2470                        bi = yaffs_GetBlockInfo(dev, dev->gcBlockFinder);
2471
2472                        pagesUsed = bi->pagesInUse - bi->softDeletions;
2473
2474                        if (bi->blockState == YAFFS_BLOCK_STATE_FULL &&
2475                                pagesUsed < dev->param.nChunksPerBlock &&
2476                                (dev->gcDirtiest < 1 || pagesUsed < dev->gcPagesInUse) &&
2477                                yaffs2_BlockNotDisqualifiedFromGC(dev, bi)) {
2478                                dev->gcDirtiest = dev->gcBlockFinder;
2479                                dev->gcPagesInUse = pagesUsed;
2480                        }
2481                }
2482
2483                if(dev->gcDirtiest > 0 && dev->gcPagesInUse <= threshold)
2484                        selected = dev->gcDirtiest;
2485        }
```

Listing 11. Function `yaffs_FindBlockForGarbageCollection` Part 2 (Excerpt fromyaffs_guts.c)

the first garbage collection after mounting of a YAFFS2 device and subsequently every 500 executions of `yaffs_FindBlockForGarbageCollection` that lead to selection of a block for garbage collection.

```
2487
2488        /*
2489         * If nothing has been selected for a while, try selecting the oldest dirty
2490         * because that's gumming up the works.
2491         */
2492
2493        if(!selected && dev->param.isYaffs2 &&
2494                dev->gcNotDone >= ( background ? 10 : 20)){
2495                yaffs2_FindOldestDirtySequence(dev);
2496                if(dev->oldestDirtyBlock > 0) {
2497                        selected = dev->oldestDirtyBlock;
2498                        dev->gcDirtiest = selected;
2499                        dev->oldestDirtyGCs++;
2500                        bi = yaffs_GetBlockInfo(dev, selected);
2501                        dev->gcPagesInUse =  bi->pagesInUse - bi->softDeletions;
2502                } else
2503                        dev->gcNotDone = 0;
2504        }
2505
2506        if(selected){
2507                T(YAFFS_TRACE_GC,
2508                  (TSTR("GC Selected block %d with %d free, prioritised:%d" TENDSTR),
2509                  selected,
2510                  dev->param.nChunksPerBlock - dev->gcPagesInUse,
2511                  prioritised));
2512
2513                dev->nGCBlocks++;
2514                if(background)
2515                        dev->backgroundGCs++;
2516
2517                dev->gcDirtiest = 0;
2518                dev->gcPagesInUse = 0;
2519                dev->gcNotDone = 0;
2520                if(dev->refreshSkip > 0)
2521                        dev->refreshSkip--;
2522        } else{
2523                dev->gcNotDone++;
2524                T(YAFFS_TRACE_GC,
2525                  (TSTR("GC none: finder %d skip %d threshold %d dirtiest %d using %d oldest %d%s"
                          TENDSTR),
2526                  dev->gcBlockFinder, dev->gcNotDone,
2527                  threshold,
2528                  dev->gcDirtiest, dev->gcPagesInUse,
2529                  dev->oldestDirtyBlock,
2530                  background ? " bg" : ""));
2531        }
2532
2533        return selected;
2534 }
```

Listing 12. Function `yaffs_FindBlockForGarbageCollection` Part 3 (Excerpt fromyaffs_guts.c)

VI. Summary

In this report we introduced YAFFS2's basic characteristics and behavior. In the first part we showed that YAFFS2 is a truly log-structured file system which makes it highly likely that parts of deleted or modified files can be found on a YAFFS2 device within a forensic investigation. We also introduced YAFFS2 way to use NAND flash memory's pages' OOB areas to organize data and analyzed discrepancies between YAFFS2's documentation and its actual behavior. As can be seen in Section III of this report, YAFFS2's documentation proved to be generally accurate but did not provide detailed information on some of YAFFS2's behavior's aspects, especially YAFFS2's garbage collection techniques.

During the analysis of YAFFS2's garbage collection techniques it became obvious that garbage collection has substantial impact on the amount of obsolete chunks containing potential evidence that can be recovered from a YAFFS2 device within a forensic analysis. As can be seen in Section IV, YAFFS2 uses sophisticated methods to decide whether garbage collection has to be performed and, if yes, which block is best suited to be garbage

```
142 __u32 yaffs2_FindRefreshBlock(yaffs_Device *dev)
143 {
144         __u32 b ;
145
146         __u32 oldest = 0;
147         __u32 oldestSequence = 0;
148
149         yaffs_BlockInfo *bi;
150
151         if(!dev->param.isYaffs2)
152                 return oldest;
153
154         /*
155          * If refresh period < 10 then refreshing is disabled.
156          */
157         if(dev->param.refreshPeriod < 10)
158                 return oldest;
159
160         /*
161          * Fix broken values.
162          */
163         if(dev->refreshSkip > dev->param.refreshPeriod)
164                 dev->refreshSkip = dev->param.refreshPeriod;
165
166         if(dev->refreshSkip > 0)
167                 return oldest;
168
169         /*
170          * Refresh skip is now zero.
171          * We'll do a refresh this time around....
172          * Update the refresh skip and find the oldest block.
173          */
174         dev->refreshSkip = dev->param.refreshPeriod;
175         dev->refreshCount++;
176         bi = dev->blockInfo;
177         for (b = dev->internalStartBlock; b <=dev->internalEndBlock; b++){
178
179                 if (bi->blockState == YAFFS_BLOCK_STATE_FULL){
180
181                         if(oldest < 1 ||
182                                 bi->sequenceNumber < oldestSequence){
183                                 oldest = b;
184                                 oldestSequence = bi->sequenceNumber;
185                         }
186                 }
187                 bi++;
188         }
189
190         if (oldest > 0) {
191                 T(YAFFS_TRACE_GC,
192                   (TSTR("GC refresh count %d selected block %d with sequenceNumber %d" TENDSTR),
193                    dev->refreshCount, oldest, oldestSequence));
194         }
195
196         return oldest;
197 }
```

Listing 13. Function `yaffs_FindRefreshBlock` (Excerpt from yaffs_yaffs2.c)

collected. Despite the complexity of YAFFS2's garbage collector's techniques to select a block for garbage collection, a basic goal of these techniques became clear, namely to select that block for garbage collection that features the largest number of obsolete chunks. Additionally, preferably older blocks are garbage collected. We discovered that YAFFS2 uses aggressive garbage collection only in case a NAND does not have enough free blocks available to store checkpoint data. The answer to the question whether garbage collection has to be performed at a specific time depends mostly on the ratio of free chunks within free blocks to the total amount of free chunks and on the kind of thread that was used to invoke a check for garbage collection. As can be seen in

Section IV, a check for necessity of garbage collection only leads to further steps of garbage collection if storage space is so scarce that aggressive garbage collection is necessary, the check was invoked by a background check or at max one quarter of all a device's free chunks reside within free blocks. A further analysis and practical evaluation of YAFFS2's garbage collection techniques and their impact on forensic analysis of YAFFS2 devices is provided by Zimmermann [8].

In the last part of this report, we analyzed YAFFS2's wear leveling techniques. As can be seen in Section V, YAFFS2 performs wear leveling mostly implicitly and uses its garbage collector for explicit wear leveling also called block refreshing. Block refreshing is at least performed during the first execution of garbage collection after a YAFFS2 device has been mounted. Additionally, block refreshing is performed on a regular basis, by default every 500 times a block is selected to be garbage collected. Block refreshing is just a variant of regular garbage collection with the crucial difference that block refreshing always selects the oldest block for garbage collection, regardless of the number of obsolete chunks within this block.

ACKNOWLEDGMENTS

This work has been supported by the Federal Ministry of Education and Research (grant 01BY1021 – MobWorm). We would also like to thank Felix Freiling for his valuable input and comments.

REFERENCES

[1] C. Manning, "Yaffs 2 specification and development notes," 2005, [Online] http://www.yaffs.net/yaffs-2-specification-and-development-notes, Last Accessed: April 22, 2011.

[2] ——, "Yaffs licensing overview." [Online]. Available: http://www.yaffs.net/yaffs-licensing-overview

[3] Aleph One Ltd., "The yaffs2 repository." [Online]. Available: http://www.aleph1.co.uk/gitweb?p=yaffs2.git;a=summary

[4] C. Manning, "How yaffs works," 2010, [Online] http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf, Last Accessed: April 22, 2011.

[5] ——, "Structure of yaffs2 spare oob area," 2011, [Email communication] http://balloonboard.org/lurker/message/20110203.214218.e77b9b11.en.html, Last Accessed: April 22, 2011.

[6] IEEE Computer Society and The Open Group, "Standard for information technology 1003.1 - portable operating system interface," 2008.

[7] C. Manning, "Time stamps in file header chunks," 2011, [Email communication] http://www.aleph1.co.uk/lurker/message/20110323.224326.cc219d84.en.html, Last Accessed: April 22, 2011.

[8] C. Zimmermann, "Mobile Phone Forensics: Analysis of the Android Filesystem (YAFFS2)," Master's thesis, University of Mannheim, 2011.

[9] T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, 2008, pp. 327–338.

[10] C. Manning, "Block refreshing," 2011, [Email communication] http://www.aleph1.co.uk/lurker/message/20110320.212508.2a671a70.en.html, Last Accessed: April 22, 2011.