# Recursive Updates in Copy-on-write File Systems - Modeling and Analysis

Jie Chen*, Jun Wang†, Zhihu Tan*, Changsheng Xie*

*School of Computer Science and Technology
Huazhong University of Science and Technology, China
*Wuhan National Laboratory for Optoelectronics, Wuhan, Hubei 430074, China
chenjie2003@hust.edu.cn, {stan, cs_xie}@hust.edu.cn

†Dept. of Electrical Engineering and Computer Science
University of Central Florida, Orlando, Florida 32826, USA
jwang@eecs.ucf.edu

***Abstract*—Copy-On-Write (COW) is a powerful technique for data protection in file systems. Unfortunately, it introduces a recursively updating problem, which leads to a side effect of write amplification. Studying the behaviors of write amplification is important for designing, choosing and optimizing the next generation file systems. However, there are many difficulties for evaluation due to the complexity of file systems. To solve this problem, we proposed a typical COW file system model based on BTRFS, verified its correctness through carefully designed experiments. By analyzing this model, we found that write amplification is greatly affected by the distributions of files being accessed, which varies from 1.1x to 4.2x. We further found that write amplification is also affected by the number of files being accessed, the number of files contained in a file system, and as well as the space utilization of file system trees.**

***Index Terms*—copy-on-write, file systems, write amplification**

## I. Introduction

Copy-On-Write (COW) is one of the fundamental update policies used when modifying data in disk blocks. With COW update policy, the target block is read into memory, modified, and then written to disk at an alternate location (not overwriting the old data). Since it never overwrites old data, COW is usually used to prevent data loss from system crashes in file systems [1-3].

Nevertheless, COW introduces an unpleasant recursive updating procedure. Assuming that a file system is a large tree made up of disk blocks, when a leaf block is modified with the COW policy, its parent node also needs to be modified to update the new location of the modified child block. This update process will recursively occur until it reaches the root block which can be updated in a fixed place on disk. We define such a procedure as a recursive update. Recursive updates can lead to several side effects to a storage system, such as write amplification (also can be referred as additional writes) [4], I/O pattern alternation [5], and performance degradation [6]. This paper focuses on the side effects of write amplification.

Studying the behaviors of write amplification is important for designing, choosing, and optimizing the next generation file systems, especially when the file systems uses a flash-memory-based underlying storage system under online transaction processing (OLTP) workloads. That's because the OLTP workloads introduce random write access pattern, which would trigger the worst case of recursive updates. Besides, the flash-memory media would also suffer from the effects of high write amplification because of its limited write-endurance and poor write performance.

There are many difficulties for evaluating the behaviors of write amplification. First, the mainstream COW file systems like ZFS [2], BTRFS [3] are running in the OS kernel. It is hard to hack these file systems for evaluation because of their complex implementation. Second, a recursive update process is affected by many factors such as the organization of a file system, the number of files contained in a file system, the distribution of files being accessed, as well as the time epoch a checkpoint lasts. It is hard to evaluate how these factors influence the write amplification in a real file system.

To solve this problem, we proposed a typical COW file system model based on BTRFS, and verified its correctness through carefully designed experiments. By analyzing this model, we found that write amplification is greatly affected by the distributions of files being accessed, which varies from 1.1x to 4.2x. We further found that write amplification is also affected by the number of files accessed, the number of files contained in the file system, and as well as the space utilization of file system trees.

The contributions of this paper are:

Manuscript received March 2, 2014; revised May 21, 2014; accepted May 22, 2014.


doi:10.4304/jcp.9.10.2342-2351

- To our knowledge, the first study to systematically analyze the behaviors of write amplification caused by recursive updates;
- Proposed a B-tree based file system model.

The rest of this paper is organized as follows. In Section II, we motivate our work by discussing the background of recursive updates. In Section III, we describe our COW file system model. Section IV describes the methodology of calculation and verification. Section V describes the verification results and theoretical analysis results. Section VI discusses the related work and Section VII concludes our work.

## II. BACKGROUND

This section provides the background of recursive updates. Here, we discuss what is copy-on-write, what is the definition of recursive updates, how does it work in file systems, and what are their effects.

### A. What Is COW?

COW is one of the fundamental update policies used in storage systems. The basic schema is never overwriting old data. When updating a block with COW policy, the data block is read into memory, modified, and then written to a new location, leaving the old data unmodified. COW update policy has been used vastly in storage systems:

*Protecting data*: File systems like WAFL [1], ZFS [2], and BTRFS [3] use COW update policy to implement snapshot for data protection.

*Improving performance*: Log-structured file systems, such as LFS [7], use COW update policy to transform the access pattern from a large amount of small random writes to a single large sequential write, which leverages the disk sequential I/O semantics.

*Updating data on special media*: Write-once-read-many media, such as optical disk [8], uses COW to implement random write. Flash-memory file systems, such as CFFS [9], FlashFS [10], JFFS [11], use COW to optimize update operations to improve write performance and implement wear-leveling.

Different than COW, the natural update policy is called Update-In-Place (UIP), which means the target data is read into memory, modified, and then written to disk at its original location (overwriting the old data).

### B. What Is the Definition of Recursive Updates?

File systems can be conceptually modeled as a tree of disk blocks, as seen in Fig. 1. The file system tree rooted at the super block. Inodes are the immediate children of the root, and they in turn are the parents of data blocks and/or indirect or even double-indirect blocks. Thus, every allocated block with the exception of the super block has a parent.

COW update policy causes recursive updates in a file system tree. In COW file systems, a modification to a disk block is always written to a newly allocated block, which recursively updates the appropriate pointers in the parent blocks. Fig. 2 illustrates this process. When the application requests to modify the block f, the block f is

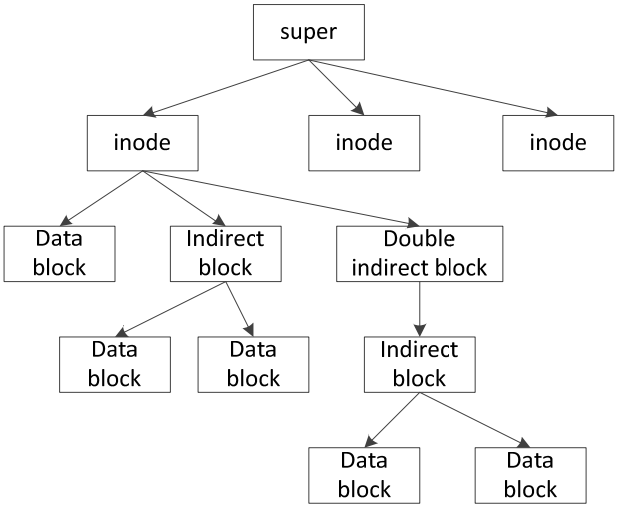


Figure 1. A file system can be conceptually modeled as a tree made up of disk blocks.

not modified directly. Instead, a new block F is allocated, and then the data in f is copied to F, and then requested modifications are made in F. However, the modified data in block F cannot be seen by the file system until its block address has been updated in its parent block d. This means that the modification to the child propagates to its parent block. Furthermore, this modification to the parent block will continue propagating along with the child-parent path until it reaches a special node which can be modified at a fixed place. We define such a procedure as a recursive update.

In order to mitigate the overhead of recursive updates, COW file systems usually use a checkpoint (or transaction) mechanism. The checkpoint (or transaction) mechanism is used to accumulate updates in memory and apply them all at once to form a consistency view of the whole file system structure. (In the following, we refer checkpoint as the operation of flushing all modified data to disk, while referring transaction as the process of accumulating updates and flushing them back during two contiguous checkpoints.) As seen from Fig. 2, after the last transaction is flushed to disk, a new transaction will start. Within the transaction, modifications to a block only trigger its COW operation once at the first time it is modified, which means a previously modified block can be updated in place. Finally, the transaction commit operation will perform a checkpoint. Several conditions can trigger the commit operation, such as a calling to *fsync*, a write-operation with *O_SYNC* flag, the number of modified blocks reaches a predefined upper limit, as well as the current transaction is timeout (usually 30s).

### C. What Are the Effects of Recursive Updates?

Here, we identify three side effects of recursive updates:

*Write amplification*: Recursive updates may cause write amplification. As shown in Fig. 2, the application only needs to modify one leaf data block f, however, the recursive update causes a total of four tree blocks (super, A, D, F) modified. So, the data actually flushed is as high as 4x of the data requested. In practice, the amount of

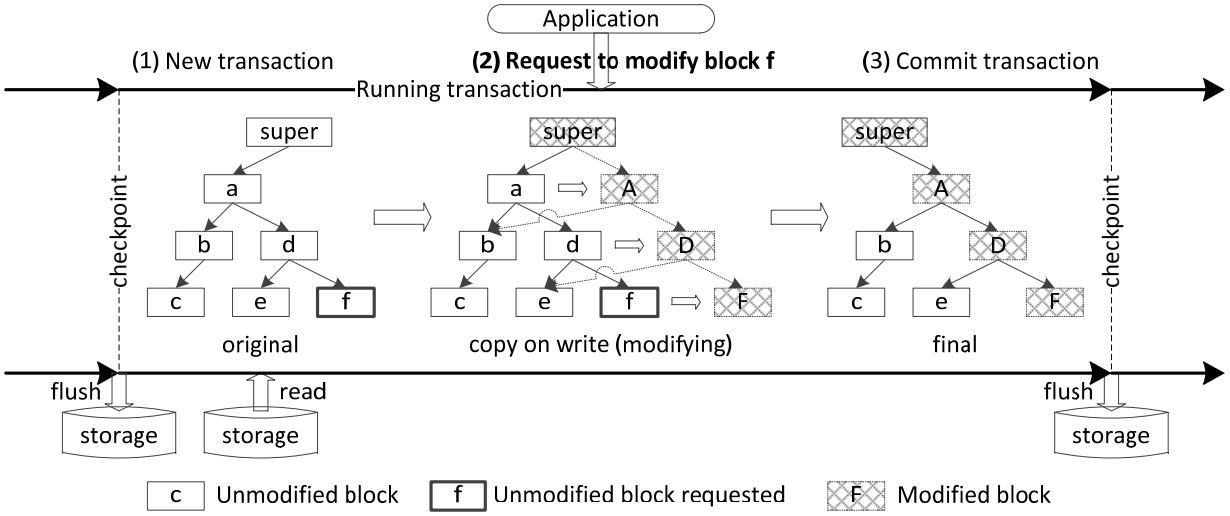


Figure 2. Recursive updates within a transaction in COW file systems.

blocks modified may be higher, since the recursive updates caused by block allocation/release are ignored in this case.

*I/O pattern alternation*: Recursive updates may change the I/O pattern at the underlying storage. As shown in Fig. 2, suppose that block f is a large contiguous data extent, while recursive updates cause additional tree blocks modified (which usually are 4KB-sized). These tree-blocks are unlikely to be contiguously stored together, thus making the I/O pattern be altered from large sequential writes to small random writes.

*Performance degradation*: The above effects eventually degrade the file system performance. Write amplification introduces additional data to write. Access pattern alternation may result in poor performance at the underlying storage.

In this paper, we only focus on the effect of write amplification.

## III. Analytical Modeling

In this section we discuss our BTRFS based file system model. First, we give a brief introduction of BTRFS and its core data structure B-tree, and then we proposal a simple model for B-tree, and then derive it to a B-tree based file system.

### A. Introduction to BTRFS

BTRFS (B-tree file system) is a GPL-licensed COW file system for Linux, which is intended to address the lack of pooling, snapshots, checksums and integral multi-device spanning in Linux file systems.

BTRFS uses COW-friendly B-trees to store metadata and extents to organize file data. A COW-friendly B-tree [12] is a variant of a standard B-tree. It adds reference counts and removes the leaf linking to the balancing algorithms of a standard B-tree. A B-tree in file systems is made up of tree blocks. Various data types are stored in such tree blocks as generic items, which are sorted by 136-bit keys. Extents are contiguous runs of disk blocks, which contain only file data.

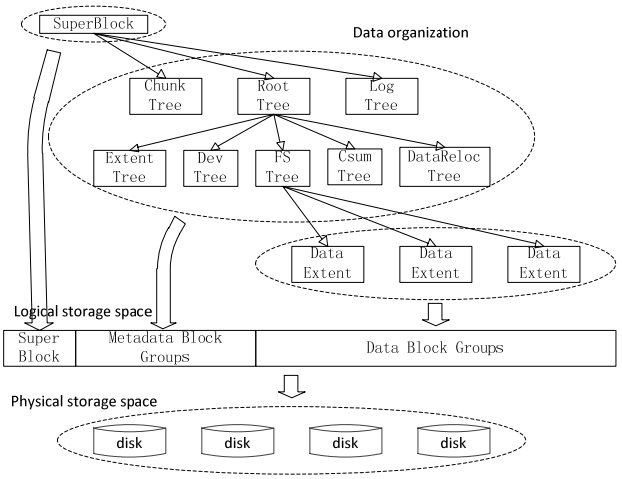


Figure 3. The overview of BTRFS. BTRFS is structured as several layers of COW-friendly B-trees.

BTRFS is structured as several layers of COW-friendly B-trees, as seen in Fig. 3. The most important trees in BTRFS are Root Tree, Extent Tree and FS Tree. Root Tree is a tree of tree roots, which stores the roots of Extent Tree, FS Tree, etc. Extent Tree tracks space allocation for both metadata and data. FS Tree contains the user-visible files and directories, which are represented by inode items. Within each directory, directory entries stored as directory items. File data are kept outside the tree in data extents, which are tracked by extent data items. The key assigning policy used in BTRFS makes all the items related to the same file be stored together.

Supposing that a COW file system like BTRFS contains N files, each of which only takes up one data block (or extent). If we need to modify x files with rewrite-4k-operation, how many additional writes it would cause? To answer this question, we first discuss a simpler question on a single B-tree with COW update mechanism, and then we derive the results to a COW file system.

### B. Recursive Updates in a Single B-tree

In this subsection we discuss how is COW-friendly B-tree organized, and what are the best and worst effects of recursive updates on such B-tree.

#### 1) COW-friendly B-trees

The B-tree discussed here is made up by individual nodes where each node takes up 4KB of disk space, see Fig. 4. There are two kinds of nodes: leaf-node containing the key-data pairs, and index-node containing the key-pointer pairs using minimum-key rule. A pointer points to the root of a sub-tree in which all the keys are greater or equal to the key in the key-pointer pair. Index-node is used to accelerate the key-data lookup process. Each node in the tree except the root-node has entries between $M$ and $2M+1$. The root-node has entries between $1$ and $2M+1$. $M$ *is* defined as the order of the B-tree.

Suppose we need to modify $k$ key-data pairs in the leaf-nodes under COW mechanism, how many tree node modifications (additional writes) would be caused due to recursive updates? The result to this question varies greatly depending on the distribution of the key-data pairs being modified as well as the number of entries in each node. Instead of discussing a generalized distribution, we will discuss two special distributions: the best case and the worst case, which would give us a boundary of the additional writes. Worst means that the corresponding distribution causes the most serious additional writes, while the best means the least additional writes. For simplicity, we also assume that the number of entries in each node is a fixed value $m$, which represents the average number of entries in a node of the B-tree.

#### 2) The Best Case

The best case of modifying $k$ key-data entries is that all the entries are compacted together in a minimum sub-tree, see Fig. 5. The number of leaf-nodes need to be modified is $\lceil k/m \rceil$, and the number in the upper level is $\lceil k/m^2 \rceil$. So on it would follow that at level $i$ there is $\lceil k/m^{d+1-i} \rceil$ nodes to be modified. Finally, the number of the tree nodes need to be modified is:

$$M_{best}(m, d, k) = \sum_{i=1}^{d} \left\lceil k/m^{d+1-i} \right\rceil \quad (1)$$

#### 3) The worst Case

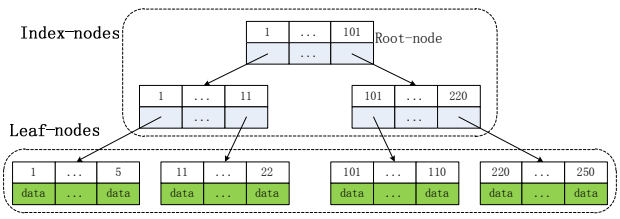


Figure 4. An example of COW-friendly B-trees. Leaf-nodes contain key-data pairs, all of which are in the same level. Index-nodes contain key-pointer pairs, which are used for accelerating the key-data pair lookup process.

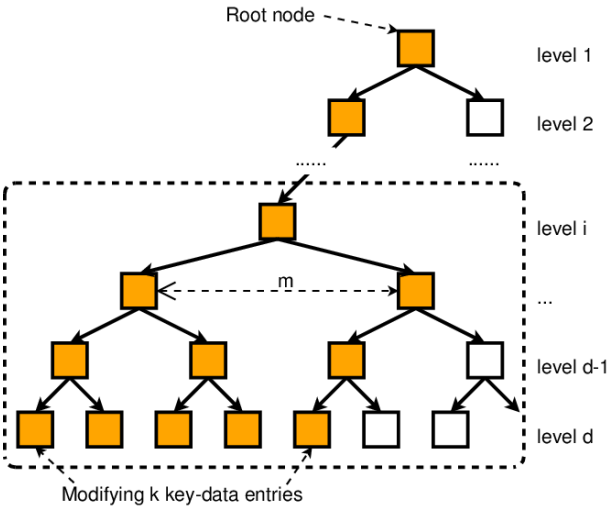


Figure 5. The best case of recursive updates. All modified $k$ key-data entries are compacted into a minimum sub-tree, then $\lceil k/m^{d+1-i} \rceil$ nodes at level i would be modified. So total nodes modified would be $\sum_{i=1}^{d} \lceil k/m^{d+1-i} \rceil$.

The worst case of modifying $k$ key-data entries is that all the entries spread out evenly, see Fig. 6, which causes $k$ nodes to be modified at each level from a level $j$+1 to level $d$, and then all nodes above level $j$+1 are modified, where $j$ should meet $m^{j-1} \le k < m^j$, that is $j = \lfloor \log_m k \rfloor + 1$. So the total nodes need to be modified is $\sum_{i=1}^{j} m^{i-1} + k*(d-j)$. Finally, the number of the tree nodes need to be modified is:

$$M_{worst}(m, d, k) = \frac{m^j - 1}{m - 1} + k * (d - j),$$
$$\text{where } j = \lfloor \log_m k \rfloor + 1 \quad (2)$$

### C. Recursive Updates in a B-tree File System

A B-tree file system is more complex than a singular B-tree. Here we model a BTRFS based COW file system for discussing.

#### 1) B-tree file system model

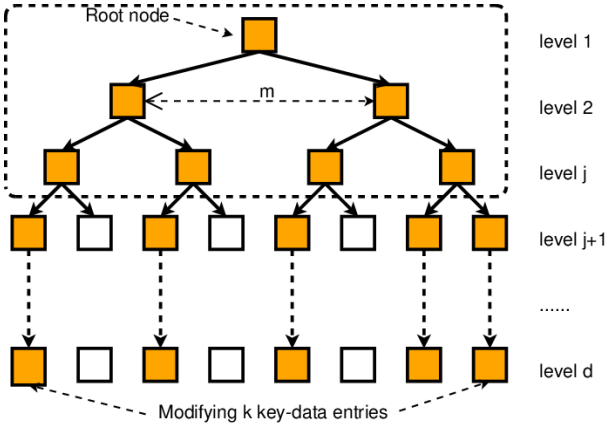


Figure 6. The worst case of recursive updates. All modified $k$ key-data entries spread out totally, which causes $k$ nodes at each level from level $d$ to a level $j+1$, where $m^{j-1} \le k < m^j$. Then all the nodes above level $j+1$ would be modified. We can resolve $j = \lfloor \log_m k \rfloor + 1$. So total nodes modified would be $\sum_{i=1}^{j} m^{i-1} + k * (d-j)$.

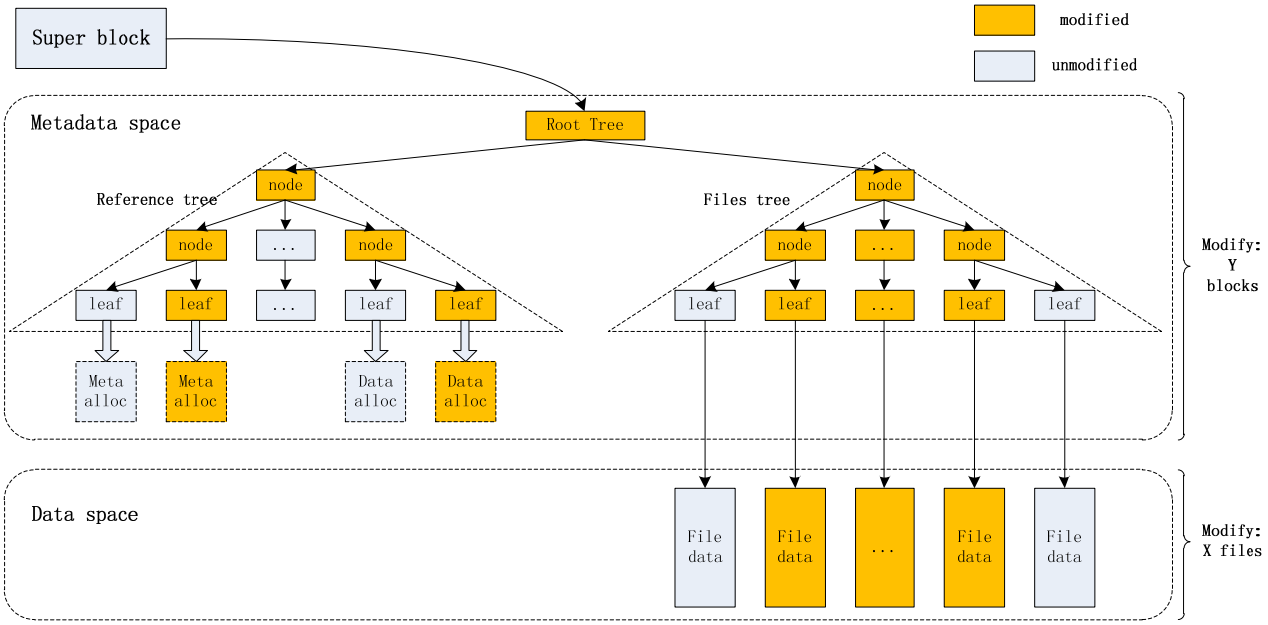


Figure 7. The B-tree file system model. Four components: super block, root-tree, reference-tree, as well as files-tree. Reference-tree records the space allocation. Files-tree records all the files metadata. Root-tree records the root of all other trees.

The modeled file system, see Fig. 7, is composed of one super-block, and three COW-friendly B-trees: root-tree, reference-tree and files-tree.

- Super-block: The super-block contains the block address of root-node of root-tree, as well as other global information in a common file system.
- Root-tree: The root-tree stores the block addresses of the roots of all other B-trees. Since the root-tree is quite small, we omit it in the later analysis.
- Reference-tree: The reference-tree stores all the reference information of disk space allocation, which includes the address information (start address and length), the reference count, as well as the owners.
- Files-tree: The files-tree stores all the file information, which includes inode and data extent mappings. Each inode represents a file, which can be a regular file or a directory file. Each file has one or more extent mappings, which maps file linear space to disk space by extents.

The modeled file system updates using COW transaction model to ensure data consistency. Its behavior is like a single B-tree's COW operation, but a little more complex because of space allocation management. When allocating or releasing space, the reference-tree would be modified, while this modification will cause new space allocation/release. This process will recursively occur until the space allocation and nodes modification get balanced in the transaction. This balance can be met because COW operation is only performed once at the first time when a node is modified in the current transaction.

The modeled file system manages the disk space allocation by dividing the space into metadata space and data space. All the tree-nodes are allocated from the metadata space, and file data extents are allocated from the data space. To improve the write performance, the file system applies lazy-allocation policy and allocates space sequentially.

*2) The depth of the trees*

Suppose there are already N files in the file system, then there would be N inodes entries and N data extent mappings entries in the leaf-nodes (assuming each file only has on extent), that is 2N entries in the files-tree.

So the depth of the files-tree $d_{files}$ would be:

$$d_{files} = \lceil \log_m(2N) \rceil \tag{3}$$

While in order to compute the depth of the reference-tree, we need to figure out how many blocks are allocated, including data blocks, file-tree blocks and reference-tree blocks. The number of data blocks is $N$. The files-tree has a number of $B_{files}$ tree blocks:

$$B_{files} = \sum_{i=1}^{d_{files}} m^{i-1} = \frac{m^{d_{files}} - 1}{m - 1} \tag{4}$$

Suppose there are $B_{ref}$ tree blocks for reference-tree, the reference-tree would have entries of $B_{ref} + B_{files} + N$. So the depth of the reference-tree $d_{ref}$ meets:

$$d_{ref} = \lceil \log_m (B_{ref} + B_{files} + N) \rceil \tag{5}$$

Meanwhile the total tree blocks of reference-tree $B_{ref}$ would be:

$$B_{ref} = \sum_{i=1}^{d_{ref}} m^{i-1} = \frac{m^{d_{ref}} - 1}{m - 1} \tag{6}$$

According to (4), (5) and (6), we can get the following $y = f(y)$ type equation (How to resolve this kind of equation refers to Appendix A):

$$d_{ref} = \left\lceil \log_m \left( \frac{m^{d_{ref}} - 1}{m - 1} + \frac{m^{d_{files}} - 1}{m - 1} + N \right) \right\rceil \tag{7}$$

From the above, when the number of files $N$ is known, we can calculate the depth of files-tree $d_{files}$ and the depth of reference-tree $d_{ref}$.

*3) The bound of additional writes*

We return to the question which is asked at the beginning of this section: Suppose there are $x$ files need to be re-written one block each, how many tree blocks would be modified? The answer to this question varies greatly depending on the distribution of $x$ accessed files. Here we discuss the worst situation and the best situation.

The worst situation is all the files being accessed are randomly selected. Re-writing $x$ files data means to modify $x$ inodes for updating time-stamps and $x$ data extent mappings for mapping new data extents, as well as allocating $x$ data extents in reference-tree for COW mechanism. Since the $x$ files random accessed as well as inodes and data extent mapping of a file are stored together, the files-tree is trigged the worst case of COW with $x$ leaf-nodes modified. So the total tree-blocks need to be modified in the files-tree are $M_{worst}(x)$, which causes the same number space allocation in the reference-tree.

Suppose there are $y_{worst}$ tree blocks need to be modified in the file system, so the reference-tree would need to modify $y_{worst}$ entries sequentially, which is the best case of COW. Besides, allocation $x$ data extents causes $x$ entries in the reference-tree sequentially modified, which is a best case too. So we get following $y = f(y)$ type equation:

$$\begin{aligned} y_{worst} &= M_{worst}(m, d_{files}, x) \\ &+ M_{best}(m, d_{ref}, x) + M_{best}(m, d_{ref}, y_{worst}) \end{aligned} \tag{8}$$

The best situation is all the files being accessed are sequentially selected, in which situation the files-tree is trigged the best case of COW with $2x$ leaf-nodes modified. And then the chain effect to the reference-tree is the same with the worst situation. Suppose there are $y_{best}$ tree blocks need to be modified in the file system, so we get following $y = f(y)$ type equation:

$$\begin{aligned} y_{best} &= M_{best}(m, d_{files}, 2x) \\ &+ M_{best}(m, d_{ref}, x) + M_{best}(m, d_{ref}, y_{best}) \end{aligned} \tag{9}$$

Finally, total tree blocks need to be modified under COW model $y_{cow}(x)$ would meet:

$$y_{best} \le y_{cow}(x) \le y_{worst} \tag{10}$$

The accurate value of $y_{cow}(x)$ is determined by the actual distribution of $x$ accessed files in the files-tree.

## IV. Methodology

This section talks about how we calculate and verify the COW file system model.

### A. Calculate the Model

From all the above equations, we can know that the overhead of recursive updates is in the range $[y_{best}, y_{worst}]$. And the range bounds are determined by the average entries in each node $m$, the number of files in the file system $N$, as well as the number of files accessed $x$.

Suppose the tree block is 4KB-sized, and each key-pointer pair takes up 32B (a key takes up 24B, and a pointer takes up 8B), then the value of $m$ should be in the range [64,128]. Yao [13] pointed that the space utilization in the B-trees is about 69%, from which we can set 128 * 69% as the reference value of $m$, that is $m$ = 88.32.

According to Meyer's research [14], the average number of files in file systems reaches about 640,000, which can be set as the reference value of $N$.

The number of files accessed in a transaction varies greatly depending on the workloads and the file system's configuration. Here, we use 100 as a reference value.

In the following, we use MATLAB to analyze the effects of above parameters on the behaviors of write amplification caused by recursive updates.

### B. Verify the Model

The basic idea of verifying the model is to measure write amplification of recursive updates in a real BTRFS under the simulated best/worst cases, and then compare them to the theoretical results.

*1) Write amplification measurement*

In order to measure the write amplification, we hacked into BTRFS to count the following two metrics: *the amount of data flushed* (denoted as $N_{data}$) and *the amount of metadata flushed* (denoted as $N_{meta}$). The two counters are reset to zero and start to count when BTRFS is mounted, the counting stops and results are written to the system log when BTRFS is dismounted. The *write amplification* (denoted as $r_{w.a.}$) discussed here is calculated as the ratio of data actually flushed by file system and the data modified by upper application during one test ($r_{w.a.} = (N_{data} + N_{meta}) / N_{data}$).

*2) Best/worst workloads simulation*

The key to simulate the best/worst workloads is to acquire relative locations of all the files in the file system tree, and then lunch modifications to the files at some special locations. To acquire the relative location, we can create predefined number of files (each file is 4KB-sized) in a blank BTRFS under the same directory, and each file is given the creating sequential number as its name. The name roughly represents the relative location of a file. The reasons are as following: (i) each file created in BTRFS has been assigned an internal id *ino*, which represents the file's creating sequence since it is allocated with the policy of last allocated *ino*+1; (ii) The FS-tree in

BTRFS stores the metadata of files in key-item manner where keys represent the locations of metadata, and the determinant part of such a key is *ino*. Suppose we need to modify *x* files, how do we select the target files? To simulate the best case, we select the *x* files sequentially from the first created file. To simulate the worst case, we select the *x* files by spreading out the *x* files evenly among all the created files.

*3) Experimental setup*

We conducted our experiments on a Dell Precision 490 workstation, which consists of 2 dual-core Intel(R) Xeon(TM) CPUs at 3.0GHz, 2GB RAM, and two internal SATA disks (One is 80GB, the other is 250GB), as shown in TABLE I. The workstation was running the Ubuntu-10.04.3 Linux distribution with kernel 2.6.32-33-generic. All of the benchmarks were executed on the internal 250GB SATA disk. To reduce the noise of other non-related applications in the experiments, we disabled as many system services as possible.

TABLE I.
EXPERIMENTAL SETUP

| Items | Description |
|---|---|
| CPU | Intel ® Xeon ™ 3.0GHz * 4 |
| RAM | 2GB |
| DISK-sys | 80GB |
| DISK-test | 250GB |
| OS | Ubuntu-10.04.3 64bit with Linux-2.6.32-33-generic |

*4) Test procedures*

In order to verify the model, we suppose there are 640,000 files in BTRFS, and the order of B-trees is 88. The number of files being accessed varies from 10 to 500 with a step of 10. All the evaluation experiments are conducted in three steps: (i) format the BTRFS and then create predefined number of initial files; (ii) remount BTRFS to reset the count to zero; (iii) run the evaluation (best/worst case workload) and read the data after dismounting BTRFS from system log files. Each experiment was conducted nine times.

## V. EVALUATION

In this section we discuss the results of verification and model analysis.

### A. Verification Results

The theoretical model simulates the behavior of write amplification in BTRFS quite well, as shown in Fig. 8. In the worst case, although the two curves of theoretical model and BTRFS show some significant differences when the number of files being accessed are in the range of [10, 50], the two curves become nearly fitting together when the number of files being accessed are in the range of [50, 500]. In the best case, the two curves of theoretical model and BTRFS show a trend of fitting together, even though the results of BTRFS is slightly higher than theoretical model. The reason of those differences is partly because the real file system BTRFS has some additional update features than our model (such as writing multi-copies of the super-block for backup), which causes some kind of fixed additional writes to the results of our model. The effect of these additional writes is significant when the number of files being accessed is small. However, it becomes negligible when the number of files being accessed is big. This indicates that our theoretical model can simulate the behavior of write amplification in BTRFS quite well, and so it's reasonable to use this model to analyze the behaviors of write amplification in BTRFS-like COW file systems.

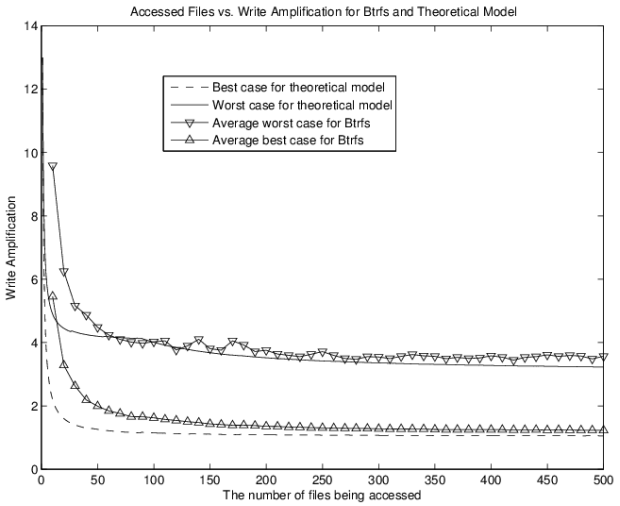


Figure 8. Accessed Files vs. Write Amplification for BTRFS and Theoretical Model

### B. Modeling Results

In the following, we discuss how the write amplification is affected by the number of files being accessed in a transaction, the number of files contained in a file system, and as well as the space utilization of B-trees.

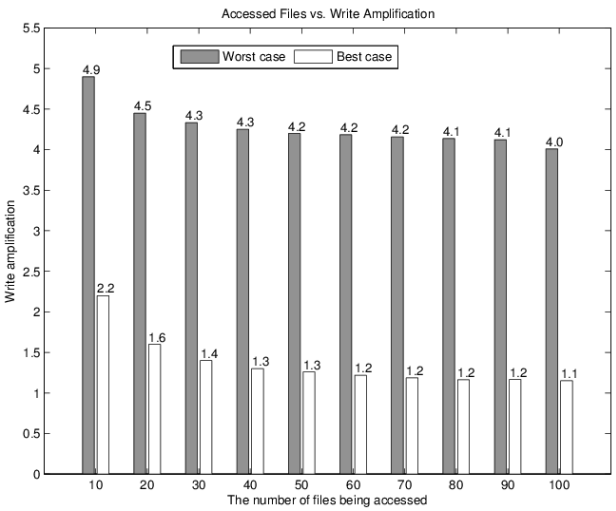


Figure 9. Write amplification impact of the number of files being accessed.

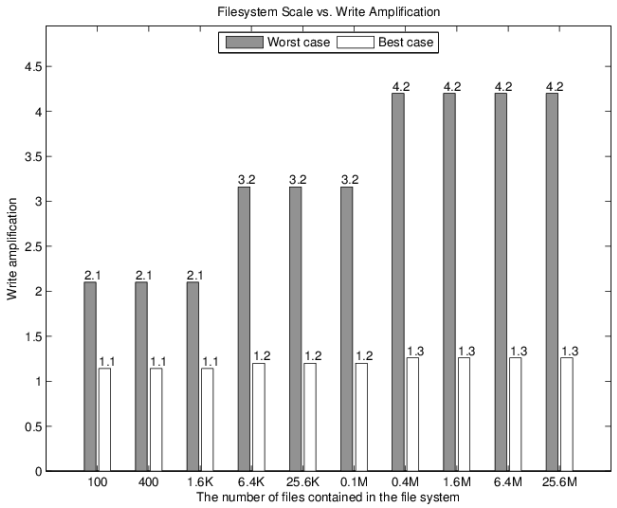


Figure 10. Write amplification impact of file system scale.

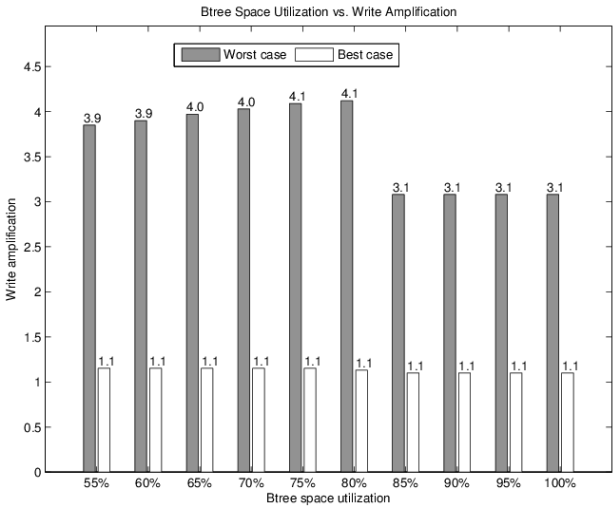


Figure 11. Write amplification impact of B-tree space utilization.

*1) Accessed files vs. write amplification*

The write amplification is reduced when the number of files being accessed is increased, as shown in Fig. 9. In the worst case, the write amplification is reduced from 4.9x to 4.0x when the number of files being accessed increased from 10 to 100. In the best case, the write amplification is reduced from 2.2x to 1.1x when the number of files being accessed increased from 10 to 100. The reason for this behavior of write amplification is that multi leaf-nodes share the same index-nodes in a file system tree, which means even though the number of leaf-nodes modified are increased, the number of related index-nodes might still remain the same. So the whole write amplification is reduced. This indicates that increasing the number of files being accessed during each transaction can reduce the effects of write amplification.

*2) Files contained vs. write amplification*

The write amplification is increased when the number of files contained in a file system is increased, as shown in Fig. 10. In the worst case, the write amplification is 2.1x when the number of files contained in a file system is from 100 to 1,600 files. And the value is increased to 3.2x when the number of files contained is from 6,400 to 100,000 files. And the value is increased to 4.2x when the number of files contained is from 400,000 to 25.6 million files. In the best case, the write amplification is 1.1x when the number of files contained in a file system is from 100 to 1,600 files. And the value is increased to 1.2x when the number of files contained is from 6,400 to 100,000 files. And the value is increased to 1.3x when the number of files contained is from 400,000 to 25.6 million files. The reason for this behavior of write amplification is because the height of a file system tree is mainly decided by the number of files contained in that file system. The more files it contains, the higher the file system tree it would be, and the more additional writes it would cause. So the whole write amplification is increased when the number of files contained in that file system is increased. This indicates that reducing the number of files contained in a file system can reduce the effects of write amplification.

*3) B-tree space utilization vs. write amplification*

The write amplification results of the best case and the worst case show different patterns when the B-tree space utilization varies from 55% to 100%, as shown in Fig. 11. In the worst case, the write amplification is increased from 3.9x to 4.1x when the space utilization is increased from 55% to 80%, while the write amplification remains the same value of 3.1x when the space utilization is increased from 85% to 100%. The reason for this behavior of write amplification is that the distribution of files being accessed is spread out evenly among the leaf-nodes of the file system tree, and the number of related index-nodes of the tree is increased as the space utilization increased while the total files being accessed remains the same, so the write amplification is increased with the increasing space utilization. While when the space utilization is increased to some threshold, the height of the tree is reduced, so the whole write amplification is reduced. In the best case, the write amplification remains the same value of 1.1x when the space utilization increased from 55% to 100%. The reason for this behavior of write amplification is because the distribution of files being accessed is aggregated at a smallest sub-tree, and even though the space utilization is increased, the related index-nodes remain nearly the same, so the whole write amplification is unchanged. This indicates that the space utilization can affect the write amplification, and improves the space utilization can significant reduce the write amplification.

## VI. Related Work

COW is used vastly in storage systems. File systems like WAFL [1], ZFS [2], and BTRFS [3] use COW mechanism to implement snapshot or clones for data protection. Log-structured file systems, such as LFS [7] use COW update policy to transform amount of small random writes to a single large sequential write to leverage the disk sequential I/O semantics. Write-once-

read-many media such as Optical Disk [8] uses COW to implement update operation. COW is also used to improve write performance and implement wear-leveling on flash-memory media both in file system level [10, 11, 15, 9] and flash translation layer level [16, 17].

The basic schema to reduce the overhead of recursive updates is the cache technique. File systems [7, 1, 2, 3] apply transaction update mechanisms to cache multiple updates in volatile memory during each transaction, in which COW operations are triggered only once on each modified block. Unfortunately, this transaction mechanism fails when encountering the synchronous writes I/O patterns, since each synchronous write requires flushing the current transaction. To avoid this circumstance, ZFS [2] provides ZIL (intent log) write cache to store a transaction group until it has safely been written to disk, which keeps the synchronous writes semantics and caches the recursive update. Reducing the size of the actual update data also can reduce the overhead of recursive updates. Inode map is the parent node of all the file inodes in the file system tree. By splitting the inode map into multiple pieces, recursive updates in LFS [7] would only cause a small subset of the whole inode map to be updated. Another way to reduce recursive updates is to find the smallest update tree of all the initial updates and make sure the recursive update stop at its root by modifying its parent update in place. This technique requires the special design of the hardware, which is implemented in byte-addressable phase-change-memory system PBFS [4]. Recursive updates also exist in nameless writes SSD systems. Zhang et al. [18] proposed segmented address space to circumvent it. The segmented address space consists of a (large) physical address space for nameless writes (like COW), and a (small) virtual address space for traditional named writes (like UIP). The basic idea is to store the metadata into virtual space, while file data into physical space. So the recursive updates triggered by modifying file data would stop at the virtual space.

## VII. Conclusions

We proposed a model to analysis the behaviors of write amplification of BTRFS-like COW file systems. We conducted real experiments on BTRFS, and verified that our model simulates the write amplification behaviors of BTRFS quite well. Through analyzing this model, we found that write amplification is greatly affected by the distributions of files being accessed, which varies from 1.1x to 4.2x. We further found that by increasing the number of files being accessed during a transaction, and reducing the number of files contained in a file system, and as well as increasing the space utilization of B-trees to some threshold, we can reduce the write amplification significantly.

## Appendix A Resolve y = f(y) Equation

From previous discussion, we can know that y is an integer, and $y \geq 0$, and $f(y) \geq y$ always to be true. So we can use following algorithm to resolve this type equation efficiently, see Algorithm 1.

**Algorithm 1** Resolve $y = f(y)$ type equation

```
Require: target function f(x)
  x ← 0
  y ← f(0)
  while x≠y do
    x ← y
    y ← f(x)
  end while
  return y
```

## Acknowledgment

We thank the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper.

This research is supported in part by the National Basic Research Program of China under Grant No.2011CB302303 and National High Technology Research Program of China under Grant No.2013AA013203, and the HUST Fund under Grant No.2014QN006. This work is also supported by Key Laboratory of Information Storage System, Ministry of Education.

## References

[1] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," in Proceedings of the USENIX Winter 1994 Technical Conference, ser. WTEC'94. Berkeley, CA, USA: USENIX Association, 1994, pp. 19–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267074.1267093

[2] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The Zettabyte File System," in FAST 2003: 2nd USENIX Conference on File and Storage Technologies, 2003.

[3] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree Filesystem," Trans. Storage, vol. 9, no. 3, pp. 9:1–9:32, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2501620.2501623

[4] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O Through Byte-addressable, Persistent Memory," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589

[5] Z. Peterson, "Data Placement for Copy-on-write Using Virtual Contiguity," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, 2002.

[6] C.-H. Wu, T.-W. Kuo, and L. P. Chang, "An Efficient B-tree Layer Implementation for Flash-memory Storage Systems," ACM Trans. Embed. Comput. Syst., vol. 6, no. 3, July 2007. [Online]. Available: http://doi.acm.org/10.1145/1275986.1275991

[7] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-structured File System," in Proceedings of the thirteenth ACM symposium on Operating systems principles, ser. SOSP '91. New York, NY, USA: ACM, 1991, pp. 1–15. [Online]. Available: http://doi.acm.org/10.1145/121132.121137

[8] J. Gait, "The Optical File Cabinet: A Random-access File System for Write-once Optical Disks," Computer, vol. 21, no. 6, pp. 11–22, June 1988. [Online]. Available: http://dx.doi.org/10.1109/2.947
[9] S.-H. Lim and K.-H. Park, "An Efficient Nand Flash File System for Flash Memory Storage," IEEE Trans. Comput., vol. 55, no. 7, pp. 906–912, July 2006. [Online]. Available: http://dx.doi.org/10.1109/TC.2006.96
[10] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," in Proceedings of the USENIX 1995 Technical Conference Proceedings, ser. TCON'95. Berkeley, CA, USA: USENIX Association, 1995, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267411.1267424
[11] D. Woodhouse, "JFFS: The Journaling Flash File System," in Ottawa Linux Symposium, vol. 2001. Citeseer, 2001.
[12] O. Rodeh, "B-trees, Shadowing, and Clones," Trans. Storage, vol. 3, no. 4, pp. 2:1–2:27, Feb. 2008. [Online]. Available: http://doi.acm.org/10.1145/1326542.1326544
[13] A. Yao, "On Random 2–3 Trees," Acta Informatica, vol. 9, no. 2, pp. 159–170, 1978.
[14] D. T. Meyer and W. J. Bolosky, "A Study of Practical De-duplication," in Proceedings of the 9th USENIX conference on File and storage technologies, 2011.
[15] C. Manning, "YAFFS: The Nand-specific Flash File System," Linuxdevices.org, 2002.
[16] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A Space-efficient Flash Translation Layer for Compact Flash Systems," Consumer Electronics, IEEE Transactions on, vol. 48, no. 2, pp. 366 –375, may 2002.
[17] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A Survey of Flash Translation Layer," J. Syst. Archit., vol. 55, no. 5-6, pp. 332–343, May 2009. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2009.03.005
[18] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for Flash-based SSDs with Nameless Writes," in Proceedings of the 10th USENIX conference on File and Storage Technologies, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208462

**Jie Chen** is a PhD student in Computer Science at Huazhong University of Science and Technology. He received the B.S. degree in Computer Science from Huazhong University of Science and Technology in 2007. His research interests include file systems, storage systems, and cloud computing.

**Jun Wang** is an Associate Professor in the Department of Electrical Engineering and Computer Science in University of Central Florida. Prior to that, he was a faculty in Computer Science and Engineering Department of University of Nebraska, Lincoln. He received his Ph.D. from University of Cincinnati in 2002. His research interests include: data-intensive high performance computing, massive storage and file system, I/O architecture, peer-to-peer system, and low-power computing.

**Zhihu Tan** is an Associate Professor in the School of Computer Science and Technology at Huazhong University of Science and Technology. He received PhD degree in Computer Architecture from Huazhong University of Science and Technology in 2008. His research interests include mass data storage, reliability of storage systems, parallel and distributed computing.

**Changsheng Xie** is the deputy Director of the Wuhan National Laboratory for Optoelectronics. He is also the Director of the Data Storage Systems Laboratory (the key laboratory of Ministry of Education of China) and the Professor in the School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests have been in the fields of networking storage system, optical and magnetic data storage technologies. He is now interested in cloud storage and big data technologies.