RJ10464 REVISED (A1004-014) April 26, 2010
Computer Science

IBM Research Report

Deferred Reference Counters for Copy-On-Write B-trees

Ohad Rodeh
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Deferred Reference Counters for Copy-On-Write B-trees

Ohad Rodeh

April 19, 2010

Abstract

Initial work on copy-on-write friendly b-trees has shown how to efficiently use reference counters to build file-systems that support snapshots and clones. However, unassisted, the disk-access pattern when accessing the ref-counters is random. This is not a good match to contemporary hard disk technology, which favors sequential workloads.

This work attempts to improve the access pattern by deferring and batching ref-count updates.

1 Introduction

The COW-btree [6] work has shown how to construct b-trees that are friendly to copy-on-write, support snapshots/clones, and employ reference-counters (*ref-counters*) to managed disk-space. Ref-counters allow the implementation of clones as first class citizens. They also allow cloning not just entire volumes but individual files as well. However, a naive implementation approach causes random accesses to the ref-count structure. Since this structure is persistently stored on disk, and does not fit in main memory, the upshot is random disk accesses. This work focuses on improving the access pattern, making it more local and sequential.

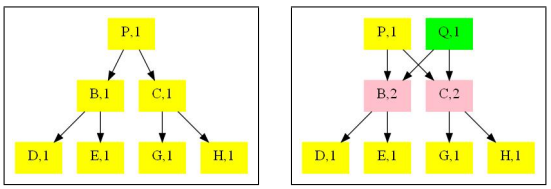
B-trees do not stand on their own, they form part of a file-system. Here, we assume a file-systm that is patterned along the lines of BTRFS [5], although, other implementations are possible. For completeness of the presentation, we sketch the important file-system algorithms here.

The file-system never overwrites blocks, block updates are aggregated in memory, and written to disk in consistency-points (CPs). The ref-counts for blocks are kept in an on-disk structure called a *ref-count tree (rc-tree)*. The rc-tree is a b-tree holding records of blocks and their ref-counts. While for simplicity we discuss b-trees that point to blocks, in reality, the file-system performs allocation in extents, and stores extents in the b-trees.

A b-tree is used to represent a file; the tree holds the mapping from file-offsets to on-disks blocks. The common file operations are read, write, create, and delete. The two operations that we shall focus on, because they have significant ref-count side-effects, are file-write and file-delete. A file-write operation is implemented as an insert-key plus data movement. A file-delete is implemented as a traversal on the tree, and ref-count reduction.

Figure 1 shows an example where tree T_p is cloned to T_q. The conventions used in the diagrams are:

- Each tree node is denoted by a letter
- The ref-count is written inside the node. In reality, the ref-counts are stored in the rc-tree, which is located in a separate on-disk structure.
- Yellow nodes are unchanged, pink nodes are unchanged excepct for their ref-counts, green nodes are newly allocated, possibly through copy-on-write.

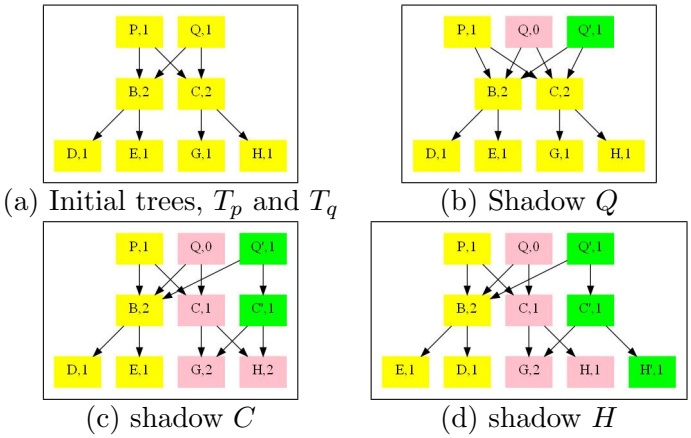


(a) Tree T_p (b) T_p cloned to T_q

Figure 1: Cloning tree T_p. A new root Q is created, initially pointing to the same blocks as the original root P. The ref-counts for the immediate children are incremented. The grand children remain untouched.

Figure 2 shows an example of an insert-key into leaf H, tree q. The nodes on the path from Q to H are $\{Q, C, H\}$. They are all modified and COWed. COWing a node involves creating a node

with the same initial content in a new location, reducing the ref-count for the original node, and incrementing the ref-counts of the children.



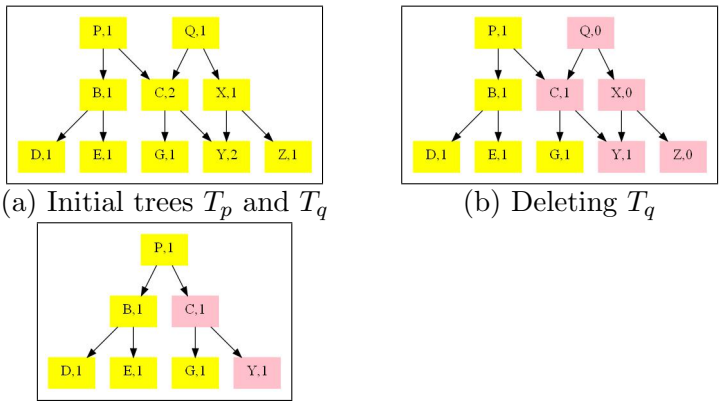
(a) Initial trees, T_p and T_q (b) Shadow Q

(c) shadow C (d) shadow H

Figure 2: Inserting a key into node H of tree T_q. The path from Q to H includes nodes $\{Q, C, H\}$, these are all COWed. Sharing is broken for nodes C and H; the ref-count for C is decremented.

Figure 3 shows an example of a tree delete. The algorithm used is a recursive tree traversal, starting at the root. For each node N:

- ref-count$(N) > 1$: Decrement the ref-count and stop downward traversal. The node is shared with other trees.
- ref-count$(N) == 1$: It belongs only to q. Continue downward traversal and deallocate N.



(a) Initial trees T_p and T_q (b) Deleting T_q

(c) Removing unallocated nodes

Figure 3: Deleting a tree rooted at node Q. Nodes $\{X, Z\}$, reachable solely from Q, are deallocated. Nodes $\{C, Y\}$, reachable also through P, have their ref-count reduced from 2 to 1.

2 Solution

There are two basic cases where ref-counts change: node-COW, and tree-delete. As each of these occur, they are translated into log-records, and stored in sequence. Processing is deferred until the next consistency point.

Some assumptions are made about how free-space is handled during a CP:

- A released block is not resued within a CP.
- Operations are admitted into a CP only if we can guaratee that there will be sufficient space for them.

This means that once the ref-count of a block reaches zero, it will not be reused withing a CP. Therefore, the changes to the ref-count of a single block are commutative; it does not matter in what order they are applied.

A copy-on-write operation from a source to a target node has the following ref-count side effects:

- Allocation of the target node, its ref-count is set to 1
- Reduction of the source node ref-count
- If the ref-count of the source is greater than 0, then the ref-counts for all the children are incremented

Figure 4 shows the two cases.

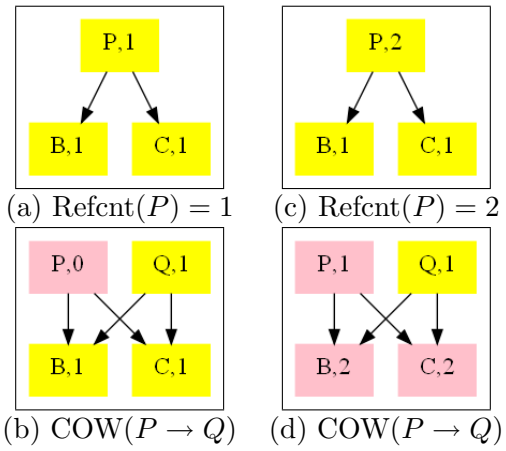
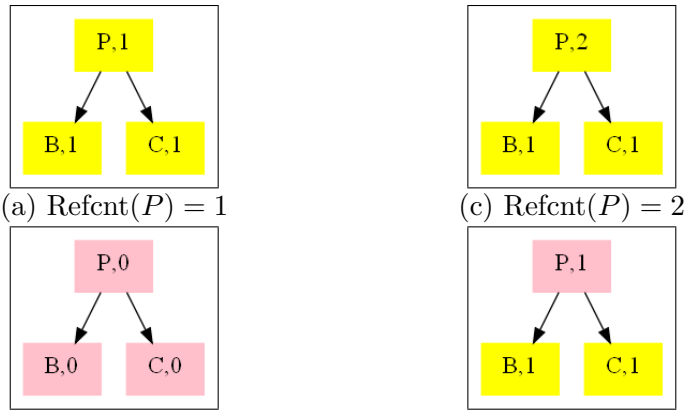


Figure 4: Node Q is a copy-on-write of node P. The left column (a,b) depicts the case where the ref-count of P is initially 1. The right column (c,d) depicts the case where the ref-count is initially greater than 1.

Deleting a tree involves reducing the ref-count of the root by one, and if the ref-count reaches zero, recursively iterating through the children. The two possible cases are shown in Figure 5.



(a) Refcnt(P) = 1 (c) Refcnt(P) = 2

(b) Traversal continues through children (d) Traversal stops at root

Figure 5: Node P is the root of a tree that is being deleted. The left column (a,b) depicts the case where the ref-count of P is initially 1. The right column (c,d) depicts the case where the ref-count is initially greater than 1.

Two log-records are used to record these changes:

- Allocate(*lba, children*): Allocate a node at address *lba*, set its ref-count to 1, and increment the ref-counts of the children.
- Decrement(*lba, children*)): Decrement the ref-count at address *lba*. If the ref-count reaches zero, then decrement the ref-count of the children, and then iterate through the children.

When node P with children c is COWed onto node Q, two log records are created: Allocate(Q, c), and Decrement(P, c). When a tree rooted at node P with children c is deleted, the log-record Decrement(P,c) is created.

As part of creating the next consistency point, the log records that have been kept are processed. A temporary in-memory structure is used to keep track of the modified ref-counts, it is called the *mem-rc-tree*. It is a tree keyed by block-address, keeping track of the modified ref-counts. There are two possible value types: (1) an absolute value (2) a difference value. For example, for block α, we might have values: {1, -2, +4}. A value of 1, means that at address α the ref-count after the CP is going to be 1. A value of -2 means that the ref-count at α should change by -2. A value of +4 means that the ref-count should be increased by 4.

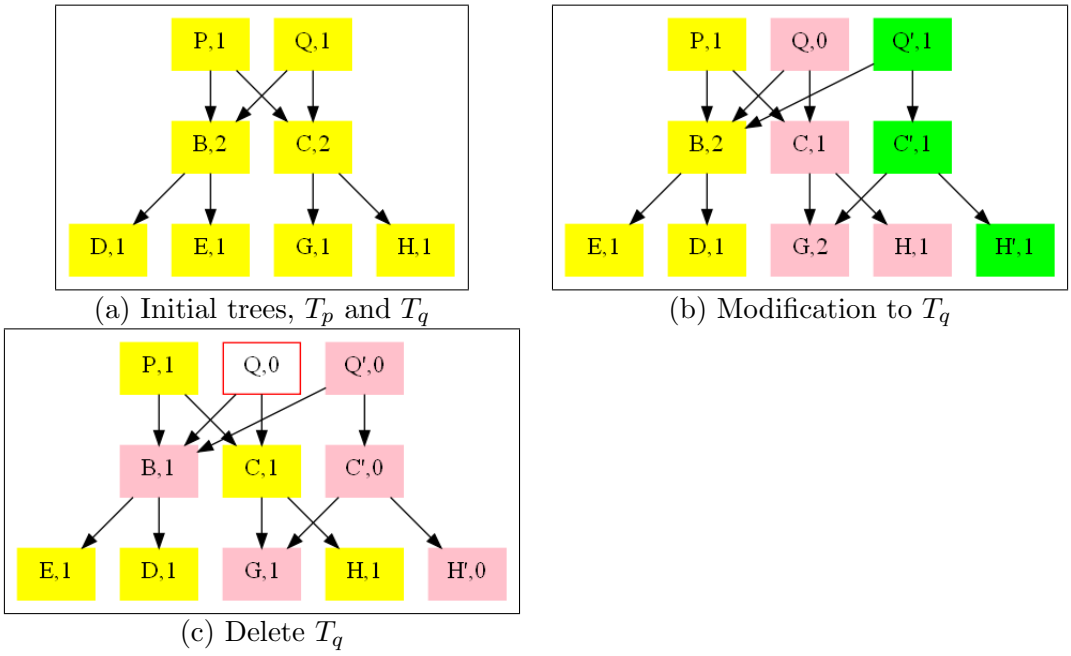
Since ordering does not matter, we sort and perform all the allocations first. These do not require any disk IO. Then, we go through the deallocations. Some of these are removals of freshly allocated blocks, and we do not need to read anything from disk. Some of them require reading ref-counts from the on-disk rc-tree. The read requests are sorted by block-address, and sent in parallel. This is done prior to actual processing. Since some ref-counts will reduce to zero, we will need to read and process the ref-counts for their children. This is a recursive procedure that can take several iterations to complete. Processing is complete once the mem-rc-tree contains only absolute or positive difference values.

The fully specified mem-rc-tree can either be written to disk as a set of log records to be applied later, or, it can be integrated with the on-disk rc-tree as part of the CP.

In terms of access pattern, the IOs at each stage are sent in parallel, in tree-sorted fashion. This is a big improvement over the basic random access pattern.

3 A detailed scenario

The algorithm reorders the ref-count operations. It is interesting to look at an example, to see that correctness is preserved. Figure 6 depicts an example where the initial state includes file p and its clone q. Later, q is updated, and some sharing is lost. Finally, q is deleted.



(a) Initial trees, T_p and T_q

(b) Modification to T_q

(c) Delete T_q

Figure 6: File p has two indirect blocks, four data blocks, and root Q. File q is a clone of p. (a) Initially, p and q share all data and metadata blocks. (b) File q is modified. (c) File q is deleted.

The log records created for steps (b) and (c), assuming they occur in the same consistency point, would be:

1. COW($Q \rightarrow Q'$)
   - Allocate($Q', \{B, C\}$)
   - Decrement($Q, \{B, C\}$)
2. COW($C \rightarrow C'$)
   - Allocate($C', \{G, H\}$)
   - Decrement($C, \{G, H\}$)
3. COW($H \rightarrow H'$)
   - Allocate($H', \emptyset$)
   - Decrement($H, \emptyset$)
4. Delete(T_q)

- Decrement$(Q', \{B, C'\})$

The logs are sorted, the Allocate records are done first, and the Decrement records are done last. This produces the following operational ordering:

1. Allocate$(Q', \{B, C\})$
2. Allocate$(C', \{G, H\})$
3. Allocate$(H', \emptyset)$
4. Decrement$(Q, \{B, C\})$
5. Decrement$(C, \{G, H\})$
6. Decrement$(H, \emptyset)$
7. Decrement$(Q', \{B, C'\})$

In what follows we annotate the entries of the the mem-rc-tree with the children of each node. This is done for clarity of the exposition, it is not done in reality. After performing the allocations, the mem-rc-tree includes the following entries:

node	children	ref-count
B	$\{E, D\}$	+1
C	$\{G, H\}$	+1
C'	$\{G, H'\}$	1
G	$\emptyset$	+1
H	$\emptyset$	+1
H'	$\emptyset$	1
Q'	$\{B, C'\}$	1

After performing the decrements, the mem-rc-tree looks like this:

node	children	ref-count
B	$\{E, D\}$	+1
C	$\{G, H\}$	+0
C'	$\{G, H'\}$	1
G	$\emptyset$	+1
H	$\emptyset$	+0
H'	$\emptyset$	1
Q	$\{B, C\}$	-1
Q'	$\{B, C'\}$	0

The ref-counts for C and H have seen a net change of zero, they are therefore removed from the table. Since Q' has reached zero ref-count, it will be deallocated, and the ref-counts for its children ($\{B, C'\}$) are decremented. This produces the following entries:

node	children	ref-count
B	$\{E, D\}$	+0
C'	$\{G, H'\}$	0
G	$\emptyset$	+1
H'	$\emptyset$	1
Q	$\{B, C\}$	-1
Q'	$\{B, C'\}$	0

Node C' has reached zero ref-count. It will be deallocated, and the ref-counts for its children ($\{G, H'\}$) are decremented. The resulting entries are:

node	children	ref-count
C'	$\{G, H'\}$	0
G	$\emptyset$	+0
H'	$\emptyset$	0
Q	$\{B, C\}$	-1
Q'	$\{B, C'\}$	0

All this has been done without issuing any disk IOs. Next, the ref-count for Q is retrieved from disk. It is 1, therefore, Q is deallocated, the ref-counts of its children ($\{B, C\}$) are decremented. The entries are now:

node	children	ref-count
B	$\{E, D\}$	-1
C	$\{G, H\}$	-1
C'	$\{G, H'\}$	0
H'	$\emptyset$	0
Q	$\{B, C\}$	0
Q'	$\{B, C'\}$	0

The ref-counts of B and C and retrieved from disk. They are both 2, resulting in the final table:

node	children	ref-count
B	$\{E, D\}$	1
C	$\{G, H\}$	1
C'	$\{G, H'\}$	0
H'	$\emptyset$	0
Q	$\{B, C\}$	0
Q'	$\{B, C'\}$	0

Only the ref-counts for blocks Q, B, and C had to be read from disk. This is the mimimum necessary.

4 Related Work

There are two file-systems other than BTRFS that employ copy-on-write as an implementation strategy. They also support clones and snapshots. These are WAFL [3, 4] and ZFS [7].

ZFS [2, 1] does not support clones as first class citizens. In order to create a clone, a snapshot of a volume must be created, and then the clone can be based off of the snapshot. The snapshot cannot be erased until the clone is erased. While this is a limitation, it allows an efficient implementation of garbage collection. A timestamp is associated with each disk block, denoting when it was created. Snapshots can essentially be assumed to occur in a linear timeline. When snapshots and files are deleted, block timestamps can be compared against the timeline of live snapshots. A simple comparison reveals whether a block is still accessible from any live snapshot, all other blocks can be deleted. For each area on disk, ZFS maintains a set of log records describing the set of allocations and deallocations. This log can be efficiently read into memory and sorted into a tree describing what areas are allocated.

WAFL, like ZFS, does not support clones as first class citizens. To create a clone from a volume requires a snapshot. WAFL uses a free-space map with a record per block. The record contains a bit per per each potential snapshot. The bit is set to 1, if the block is a member of the snapshot. This means that there are no dependencies between a father and its children, as in the ref-count scheme. The cost is wide records. For exapmle, for 256 potential snapshots, a byte per block is required in the free-space map. In order to avoid the random access pattern to the free-space map, free-space changes are logged and written to disk. The log is applied to a particular free-space page only when the number of logged changes exceeded a threshold. This allows reading the page and applying the changes en-mass.

5 Summary

Ref-counts are a method for keeping track of space utilization in a file-systems that supports clones and snapshots. This work has shown how to improve the access pattern to the ref-counters.

6 Acknowledgments

The author would like to thank Chris Mason for describing the BTRFS implementation, and how it has worked around some of these ref-count performance issues.

References

[1] M. Ahren. http://blogs.sun.com/ahrens/entry/is_it_magic.

[2] J. Bonwick. http://blogs.sun.com/bonwick/entry/space_maps.

[3] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *USENIX*, 1994.

[4] J. Edwards, D., Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. Smith, and E. Zayas. Flexvol: flexible, efficient file volume virtualization in wafl. In *USENIX Annual Technical Conference*, pages 129–142, Berkeley, CA, USA, 2008. USENIX Association.

[5] C. Mason. BTRFS. http://en.wikipedia.org/wiki/Btrfs.

[6] O. Rodeh. B-trees, Shadowing, and Clones. *ACM Transactions on Storage*, Feb 2008.

[7] V. Henson, M. Ahrens, and J. Bonwick. Automatic Performance Tuning in the Zettabyte File System. In *File and Storage Technologies (FAST), work in progress report*, 2003.