Versioned Pointers in Tux3

Stefan Rijkers
Student number: 0553334
s.rijkers@student.tue.nl

Mohammed El-Kebir
Student number: 0554141
m.el-kebir@student.tue.nl

Martin Harwig
Student number: 0548455
m.p.harwig@student.tue.nl

April 11, 2009

Contents

1 Introduction

Tux3 is a B-tree based versioning file system. Versioning takes place in three places: in the atime btree, extents in a file data btree and attributes in the inode table. This is achieved by using a method invented by Philips called versioned pointers. In this report this method is described in a formal way. For every operation the pseudo code is given, together with the running time analysis. Note that the description and the definitions are our own work. For every operation we try to be as precise and correct as possible.

2 Preliminaries

Snapshots are used often in industry as an efficient way of backing up data. A snapshot creates a copy of the volume data at a particular point in time. A full back-up is time consuming as the whole data must be copied. On the other hand a snapshot copy can be done instantly; so the original data continues to be available without interruption. Also, the data in snapshot is made available for future access [Gar06]. There are two common implementations of the creation of snapshots:

1. Copy-on-write
   When a snapshot is created only the meta-data about where the snapshot is stored is copied. As soon as a write occurs on the original data, the data being written to is copied to the snapshot store and the pointer in the snapshot is updated. Afterwards the initial write action is performed.

2. Redirect-on-wite
   Similarly to the previous method only the meta-data is copied upon snapshot creation. This time, however, upon a write operation the address written to is redirected to some free space. Subsequently the pointer in the current version is redirected to the address of the newly allocated free space. So in comparison with the previous method one write is saved.

The main contribution of versioned pointers is to support the operations on snapshots more efficiently.

Formally, a *snapshot* is a version of the volume data. The *origin* is the ‘head revision’ of the volume data. At any time a *snapshot* can be taken from the origin. A snapshot is identified via a *snapshot tag*. Via this id the snapshot can be accessed and operated upon.

The volume data is divided in *chunks*. Two types of chunks can be distinguished, namely *orgin chunks* and *snapshot chunks*. As the name suggests, origin chunks are part of the origin. Snapshot chunks do not occur in the origin. In an efficient implementation of a snapshot facility, identical chunks are shared among different snapshots. That’s why chunks have two types of addresses. The physical location of a chunk on the volume disk is given by the physical address of that chunk. In a snapshot, however, logical addresses are used. Every snapshot has the same set of logical addresses.

The way reuse of chunks used to be supported is by storing the mapping in a B-tree indexed by the logical chunk address. For each logical chunk address a list of exceptions of that chunk is stored. Each exception is a pair of a physical address and a bitmap indicating which other snapshots share the same chunk [Phi07].

Philips stores the mapping in a different way: he suggests to base the mapping on snapshots instead of logical addresses, inspired by revision control systems. Furthermore Philips allows snapshot to be writable. Read-write snapshots are frequently used in virtualization, sandboxing and virtual hosting setups because of their usefulness in managing changes to large sets of files [Wik09].

3 Versioned pointers

The reason Philips calls this method versioned pointers is because in a sense logical addresses are pointers. Across different snapshots the same logical address may point to a different chunk, hence the name.

Versioning information is stored in a *version tree*. The origin corresponds to the root of the version tree. On the other hand, regular snapshots correspond to the non-root nodes of this tree. Changing the origin results in a new root being created, and thus the old root becoming a regular snapshot. Whereas modifying a regular snapshot node v, results in a new node being created with as parent v. Later on we will see that a node does not necessarily correspond to a snapshot. Therefore new ids are needed: every node in the version tree is identified by a *version label*.

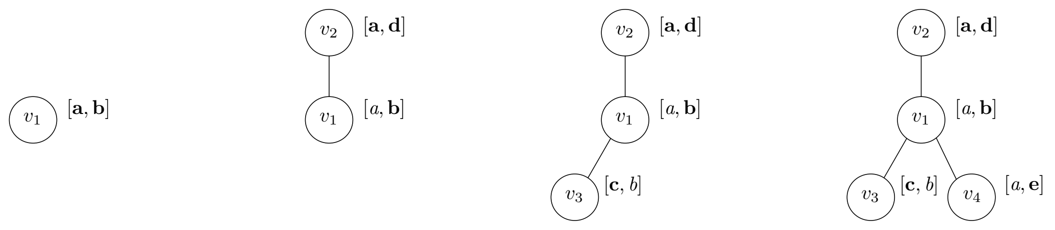
In order to accommodate reuse of chunks, an exception list $\mathcal{L}[l] = \{(v_1, p_1), \ldots, (v_k, p_k)\}$ is maintained for each logical address l. The exception list $\mathcal{L}[l]$ defines a partitioning on the nodes of the version tree $\mathcal{T}$: nodes belonging to the same partition have the same physical address for logical address l. An exception is a pair (v, p) where v is a node in $\mathcal{T}$and p is the corresponding physical address. Every node v' belonging to the subtree rooted at v and for which there is no exception on the path between v' and v belongs to the same partition as v.

It is required that in $\mathcal{L}[l]$ a physical address occurs at most once.[1] Therefore $\mathcal{L}[l]$ subdivides $\mathcal{T}$ in $k+1$ partitions. Chunks belonging to a partition of size one are called *unique*, whereas chunks in partitions of size greater than one are called *shared*. Due to the partitioning induced by an exception list, version inheritance is achieved (cf. Figure 1).

The following operations need to be supported.

1. Create snapshot of origin
2. Create snapshot of snapshot
3. Read from snapshot
4. Delete snapshot
5. Write to origin
6. Write to snapshot

[1]Suppose that $(v_1, .), (v_2, .) \in \mathcal{L}[l]$. $\mathcal{L}[l]$ would contain redundant exceptions if either v_2 belongs to the subtree of v_1 or vice versa. If the same physical chunks occurs in two different unrelated parts of $\mathcal{T}$ this would introduce inconsistency.



(a) There are no snapshots. Exception list: $\mathcal{L}[l_1] = \{(v_1, a)\}$ and $\mathcal{L}[l_2] = \{(v_1, b)\}$

(b) A write is performed to the second chunk, thus v_1 becomes a snapshot. Exception list: $\mathcal{L}[l_1] = \{(v_2, a)\}$ and $\mathcal{L}[l_2] = \{(v_1, b), (v_2, d)\}$

(c) A write to the first chunk of snapshot v_1 is performed, creating a new snapshot v_3. Exception list: $\mathcal{L}[l_1] = \{(v_2, a), (v_3, c)\}$ and $\mathcal{L}[l_2] = \{(v_1, b), (v_2, d)\}$

(d) A write to the second chunk of snapshot v_1 is performed, creating a new snapshot v_4. Exception list: $\mathcal{L}[l_1] = \{(v_2, a), (v_3, c)\}$ and $\mathcal{L}[l_2] = \{(v_1, b), (v_2, d), (v_4, e)\}$

Figure 1: An example of the described concepts, there are two logical chunks per version. Shared chunks are italic, whereas non-shared chunks are bold.

In the subsequent sections the operations are described together with their running time. In the running time, v denotes the number of nodes in $\mathcal{T}$, e denotes the maximal number of exceptions for any logical address and l denotes the number of logical chunks. It holds that $e = O(v)$, as there can be at most v exceptions for any logical address.

Chunks reside in two different stores, namely the snapshot store and the origin store. The origin store contains only the chunks used in the origin, whereas the snapshot store contains the chunks used in the snapshot.

We start by defining a node.

Definition 1 *A node $v \in \mathcal{T}$ is defined as*

1. *v.id*
   The version label of v
2. *v.depth*
   The depth of v
3. *v.parent*
   The parent of v. Note that in case v is the root then $v.\text{parent} = \textbf{nil}$
4. *v.children*
   A list of children of v, may be **nil**
5. *v.ancestors*
   A list of ancestors of v, may be **nil**

Definition 2 *The partial function f maps a tag to a node in $\mathcal{T}$, whereas the partial function g maps a node to a tag.*

3.1 Create snapshot of origin

Let r be the root node of $\mathcal{T}$. Upon creation of snapshot of r, a child node r' of r is created containing no exceptions. If r had a child, then this child is redirected to r'.

Algorithm 1: CreateOriginSnapshot($\mathcal{T}$)

```
1  Let $r$ be the root node of $\mathcal{T}$
2  Create a new node $r'$
3  **if** $|r.\text{children}| = 1$ **then**
4      Let $v$ be the child of $r$
5      $r'.\text{children} \leftarrow v$
6      $v.\text{parent} \leftarrow r'$
7      **foreach** *node $v'$ in the subtree rooted at $v$* **do**
8          $v'.\text{ancestors} \leftarrow v'.\text{ancestors} \cup r'$
9          $v'.\text{depth} \leftarrow v'.\text{depth} + 1$
10 $r.\text{children} \leftarrow r'$
11 $r'.\text{parent} \leftarrow r$
12 $r'.\text{depth} \leftarrow 1$
```

In Algorithm 1 the pseudo code is given. The running time of CreateOriginSnapshot is dominated by the update of all the ancestor lists of the nodes in $\mathcal{T}$ and thus requires $O(v)$ time.

3.2 Create snapshot of snapshot

Let v be the node for which a snapshot is created. Note that v is not the root of $\mathcal{T}$. A new child v' of v is created. The exception lists are not altered, therefore v' does not define new exceptions.

Algorithm 2: CreateSnapshot($\mathcal{T}$,v)

```
   **Input**: A node $v \in \mathcal{T}$
1  Create a new node $v'$
2  $v.\text{children} \leftarrow v.\text{children} \cup v'$
3  $v'.\text{ancestors} \leftarrow v'.\text{ancestors} \cup v$
4  $v'.\text{parent} \leftarrow v$
5  $v'.\text{depth} \leftarrow v.\text{depth} + 1$
```

In Algorithm 2 the pseudo code is given. Clearly CreateSnapshot requires $O(1)$ time.

3.3 Read from snapshot

In this operation, for a given snapshot tag t and logical address l, we need to find the corresponding physical address. This is done by scanning the exception list. Let v be the node corresponding to tag t. The goal is to find the lowest ancestral node v' such that $(v', .) \in \mathcal{L}[l]$.

Algorithm 3: FINDEXCEPTION($\mathcal{T}$,v,l)

```
Input: A node $v \in \mathcal{T}$ and a logical address $l$
Result: An exception that defines the corresponding physical for logical address $l$
1 $depth \leftarrow -\infty$
2 $v'' \leftarrow$ nil
3 $p \leftarrow$ nil
4 foreach exception $(v', p') \in \mathcal{L}[l]$ do
5     if $v' \in v$.ancestors and $v'$.depth $> depth$ then
6         $depth \leftarrow v'$.depth
7         $v'' \leftarrow v'$
8         $p \leftarrow p'$
9 return $(v'', p)$
```

Algorithm 4: READFROMSNAPSHOT($\mathcal{T}$,t,l)

```
Input: A snapshot tag $t$ and a logical address $l$
Result: A physical address
1 $v \leftarrow f(t)$
2 $(v', p) \leftarrow$ FINDEXCEPTION$(\mathcal{T}, v, l)$
3 return $p$
```

The pseudo code is given in Algorithm 4. This operation can be implemented in $O(e)$ time, provided that the membership test in Line 5 (in Algorithm 3) can be done in constant time (e.g. by using a hash table).

3.4 Delete snapshot

The case of deleting a snapshot that is a leaf is trivial. The same holds for deleting a snapshot with only one child: its exceptions can be passed on to its child (called *collapsing*). The case of a node with more than one child requires some work. The naive solution would be to propagate all its exceptions to its children. This, however, can result in more space consumption after a deletion, which of course is unacceptable. A straightforward solution is to postpone the deletion and hide the node in question. This corresponds to turning the node into a so called *ghost node*. A ghost node has a version tag but no snapshot tag, therefore it is not visible from the outside. For a ghost node the following important invariant holds.

Invariant 3 *Every ghost has at least two children.*

The question that now arises is if the number of ghosts in a version tree is bounded in terms of the number of snapshots in that tree. When doing the following observation, it's not hard to see that this is indeed the case.

Observation 4 *A leaf can never bep a ghost. Indeed, if a ghost is a leaf it is superfluous and has no right to exist.*

The following lemma now follows.

Lemma 5 *Let n be the number of snapshots. In the worst case the number of ghost is $n-1$.*

Proof. Due to Observation 4, in the worst case we have a version tree with only snapshots in the leaves. Let n be the number of leaves. We now want to bound the number of inner nodes. Invariant 3 enforces that a ghost must have at least two children. So in the worst-case each ghost node has exactly two children. Therefore the version tree is a binary tree with n leaves and thus has $n-1$ inner nodes being ghosts. $\square$

An exception (v, p) can become superfluous if none of the non-ghost nodes in the subtree rooted at v inherit that exception. Superfluous exceptions waste storage and increase the running time of the operations. So we do not want them, therefore the following invariant is defined.

Invariant 6 *For an exception (v, p) it holds that there is a non-ghost node v' in the subtree rooted at v having no exception on p, i.e. v' inherits exception (v, p).*

Invariant 6 can be violated by the operations: write to snapshot and delete snapshot. Let v be the node on which one of the two operations is performed and let l be the logical address that due to its alteration/deletion may violate the invariant. For restoring the invariant the following procedure is defined.

```
Algorithm 5: MaintainExceptionInvariant($\mathcal{T}$,$v$,$l$)
  Input: A node $v \in \mathcal{T}$ and a logical address $l$
1 $(v', p) \leftarrow$ FindException($v, l$)
2 if $v'$ is a ghost node then
3     Let $V$ be the set of non-ghost nodes in the subtree rooted at $v'$
4     if there is no exception $(w, q) \in \mathcal{L}[l]$ occurring in $V$ then
5         $\mathcal{L}[l] \leftarrow \mathcal{L}[l] - (v', p)$ ;          /* exception $(v', p)$ is superfluous */
```

In Algorithm 5 first the exception (v', p)—defining the physical address of logical address l of node v—is found in $O(e)$ time. If v' is not a ghost, then the found exception is certainly not superfluous and the invariant is not violated. If v' is a ghost, it is checked, in $O(v)$ time, whether the invariant still holds[2]. If not the exception is deleted. So the total running time of MaintainExceptionInvariant is $O(v)$.

In Algorithm 6 the three cases previously mentioned are given in pseudo code. Let v be the node to be deleted. The case where v has only one child is handled by Algorithm 7. Note that due to the deletion of v, the parent u of v may end up having only one child. If u were to be a ghost node then u must be collapsed. The running time of Collapse is $O(v \cdot l)$.[3] The same running time can be deduced for DeleteSnapshot.

[2]Again we assume that membership testing can be done in $O(1)$ time.

[3]Actually it is $O(e \cdot l + v)$

Algorithm 6: DeleteSnapshot($\mathcal{T}, t$)

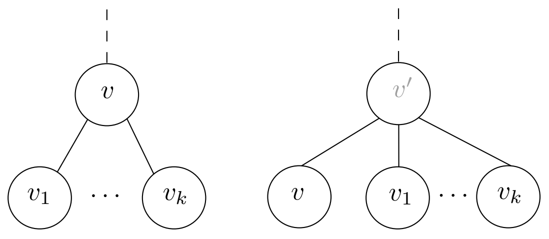
```
Input: A snapshot tag $t$
1  $v \leftarrow f(t)$
2  Let $u$ be the parent of $v$
3  if $|v.\text{children}| = 0$ then
4      Make $v$ a ghost, remove $f(g(v))$ and also remove $g(v)$
5      foreach logical address $l$ do
           /* before $v$ goes to heaven it must make peace with its ancestors */
6          MaintainExceptionInvariant($\mathcal{T}, v, l$)
7      $u.\text{children} \leftarrow u.\text{children} \setminus v$
8      Delete $v$
9  else if $|v.\text{children}| = 1$ then
10     Make $v$ a ghost, remove $f(g(v))$ and also remove $g(v)$
11     Collapse($\mathcal{T}, v$)
12 else
13     Make $v$ a ghost, remove $f(g(v))$ and also remove $g(v)$
14 if $|u.\text{children}| = 1$ and $u$ is a ghost then
       /* by Invariant 3 $u$ must be removed                          */
15     Collapse($\mathcal{T}, u$)
```

Algorithm 7: Collapse($\mathcal{T}, v$)

```
Input: A ghost node $v$ having only one child
1  Let $u, w$ be the parent and child of $v$ respectively
2  foreach logical address $l$ do
3      $(x, p) \leftarrow$FindException($\mathcal{T}, w, l$)
4      if $x = v$ then
           /* $(x, p)$ is defined at node $v$ but not overruled in node $w$, so it
              must be relabeled                                          */
5          $\mathcal{L}[l] \leftarrow (\mathcal{L}[l] \setminus (v, p)) \cup (w, p)$
6  $u.\text{children} \leftarrow (u.\text{children} \setminus v) \cup w$
7  $w.\text{parent} \leftarrow u$
8  Delete $v$
9  Fix depths and ancestor lists by performing a tree traversal on the subtree rooted at $w$
```

3.5 Write to snapshot

Writing to a snapshot is problematic due to the inheritance of exceptions. Suppose for instance that our version tree is as in Figure 1(d). Now suppose that we want to write to the second chunk of v_1, say that p becomes the new physical address of this chunk. As a consequence, $\mathcal{L}[l_2]$ is updated and no longer contains (v_1, b). This however would invalidate v_4, as now the second chunk of this node also has physical address p. As a remedy, v is turned into a *ghost node* is added that preserves the exceptions that v initially had. Subsequently, a new child node v' of v is created, see Figure 2). The snapshot tag that corresponded initially to v is remapped to v'. Also, a new exception (v', p) is added to $\mathcal{L}[l_2]$. Note that in case the node written to is a leaf then no ghost node is introduced. Observe that the node that is turned into a ghost has at least two children. Therefore Invariant 3 is not violated. The pseudo code is given in Algorithm 8. It can be seen that the running of this procedure is $O(v)$.



(a) Node v is about to be written to

(b) A new ghost node v' is introduced

Figure 2: Introduction of a ghost, note that the ghost node v' is grayed out because it is not accessible via a snapshot tag.

Algorithm 8: WriteSnapshot($\mathcal{T}$,t,l)

```
   Input: A snapshot tag $t$, a logical address $l$
1  Let $p$ be a free physical chunk in the which the data is written to
2  $v \leftarrow f(t)$
3  if $|v.\text{children}| > 0$ then
4      Create a new child node $v'$ of $v$
5      $v'.\text{ancestors} = v.\text{ancestors} \cup v$
6      $v'.\text{depth} = v.\text{depth} + 1$
7      $\mathcal{L}[l] \leftarrow \mathcal{L}[l] + (v', p)$
8      Remap $f(t)$ to $v'$ and unmap $g(v)$ and $g(v')$ to $t$
9      Make $v$ a ghost
10     MaintainExceptionInvariant($\mathcal{T}, v, l$)
11 else
12     if $(v, .) \in \mathcal{L}[l]$ then
13         $\mathcal{L}[l] \leftarrow \mathcal{L}[l] \setminus (v, .)$
14     $\mathcal{L}[l] \leftarrow \mathcal{L}[l] \cup (v, p)$
```

3.6 Write to origin

Let l be the logical chunk that is written to and let r be the root of $\mathcal{T}$. By definition we have $(r, p) \in \mathcal{L}[l]$. There are three cases to be distinguished.

1. r has no children

2. r has one child v and v occurs in $\mathcal{L}[l]$

3. r has one child v and v does not occur in $\mathcal{L}[l]$

In the first two cases nothing has to be done, except writing the data to physical address p. For the last case a new chunk with physical address p' is allocated in the snapshot store. Subsequently the data at p is copied to p' and an exception (v, p') is added to $\mathcal{L}[l]$. Finally, the data is written to p. Note that this corresponds to the copy-on-write paradigm, mentioned in Section 2.

Algorithm 9: WriteOrigin($\mathcal{T}$,l)

```
   Input: A logical address $l$
1  Let $r$ be the root of $\mathcal{T}$ if $|v.\text{children}| = 0$ then
2      F
3  ind $(r, p) \in \mathcal{L}[l]$
4  $v \leftarrow f(t)$
5  if $|v.\text{children}| = 0$ then
6      Write data to $p$
7  else
8      Let $v$ be the child of $r$
9      if $(v, .) \in \mathcal{L}[l]$ then
10         Write data to $p$
11     else
12         Allocate a new chunk $p'$ in the snapshot store
13         Copy the data in $p$ to $p'$
14         $\mathcal{L}[l] \leftarrow \mathcal{L}[l] \cup (v, p')$
15         Write the data to $p$
```

In Algorithm 9 the pseudo code is given. It can be observed that the running time is $O(e)$.

4 Discussion and Conclusion

An overview of the running times of the operations is given in Table 1.

Operation	Algorithm	Running time
CreateOriginSnapshot	(see Algorithm 1)	$O(v)$
CreateSnapshot	(see Algorithm 2)	$O(1)$
ReadFromSnapshot	(see Algorithm 4)	$O(e)$
DeleteFromSnapshot	(see Algorithm 6)	$O(vl)$
WriteSnapshot	(see Algorithm 8)	$O(v)$
WriteOrigin	(see Algorithm 9)	$O(e)$

Table 1: Operations with running times

Somehow Philips managed to implement CreateSnapshot in $O(v)$ time [Phi08a, Phi08b]. The claim by Philips that DeleteFromSnapshot runs in $O(v)$ is wrong. Apparently Philips based this claim on his test implementation that supports only one logical chunk (i.e. $l = 1$). Philips also claims that his ReadFromSnapshot operation runs in $O(e)$ time. Unfortunately, this is not correct as Philips decided to postpone fixing the ancestor lists of nodes invalidated during a delete to this operation. Therefore in the worst case, in his implementation this operation takes $O(v)$ time.[4]

A statement is made in [Phi08a] about maintaining Invariant 6 in $O(e)$ time. As as consequence according to Philips both writing to and deleting a snapshot would be possible in $O(e)$ time. This corollary is not true, as deleting a node v results in invalid depth and ancestor attributes in the nodes of the subtree rooted at v. Fixing these requires $O(v)$ time.[5]

In this report we took a step back from the actual implementation and looked at the versioned pointers technique on a more abstract level. We presented a formal definition of the concepts used. Furthermore, we defined the operations as simple as possible and proved their correctness. In retrospect, the operations do not have to be as complex as Philips' describes in his notes. Also the extensive case analysis done by Philips in various operations unnecessarily complicates things.

References

[Gar06] Neeta Garimella. Understanding and exploiting snapshot technology for data protection, part 1: Snapshot technology overview. 2006.

[Phi07] Daniel Philips. Zumastor linux storage server. In *Proceedings of the Linux Symposium*, pages 135–144, 2007.

[Phi08a] Daniel Philips. Improved versioned pointer algorithms. 2008. [Online; accessed 11-April-2009].

[Phi08b] Daniel Philips. Versioned pointers: a new method of representing snapshots. 2008. [Online; accessed 11-April-2009].

[Wik09] Wikipedia. Snapshot (computer storage) — wikipedia, the free encyclopedia, 2009. [Online; accessed 21-March-2009].

[4]In `lookup_chunk` a traversal to the root node is performed in case the node to be read has been invalidated.[Phi08a]

[5]In Philips' fuzzy test code this corresponds to the function `invalidate_path`, invoked in `promote_child` which is in turn invoked in `snapshot_delete`.[Phi08a]