

# Versioned pointers: a new method of representing snapshots

---

*Source:* <http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-07/msg02663.html>

---

- *From:* Daniel Phillips <[phillips@xxxxxxxx](mailto:phillips@xxxxxxxx)>
  - *Date:* Mon, 7 Jul 2008 03:36:47 -0700
- 

Good day storage mavens,

Every now and then down in the unexciting and arcane underbell of storage research one encounters some real computer science. Today I will outline one such encounter, which I expect to result in significant improvements to the capability and performance of our ddsnap snapshotting block device, and most probably has benefits in other areas as well, such as snapshotting filesystems. I apologize in advance for any forward references, as I wish to get straight to the meat of this. For definitions of various domain specific terms, please see the terminology section at the end.

A snapshot of a disk volume may be represented as a list of exceptions to an origin volume, where each exception points to a logical chunk of data in a "snapshot store", which holds the data for the snapshot at a particular logical address. For any logical address for which no exception is present, the data resides on the origin volume. Thus, the entire set of exceptions for a snapshot can be thought of as a list of places where the snapshot differs from the origin.

The "versioned pointer" method is a new way of representing exceptions for multiple simultaneous snapshots in a surprisingly compact form. The idea was inspired by a proposal from Steve Vandebogart (interning at Google) to represent exception sharing for a series of volume snapshots using a snapshot sequence number in place of a fixed size chunk sharing bitmap, as is the existing practice in ddsnap. The sequence number would be fewer bits than the bitmap and could be packed into unused bits of a physical block pointer, which would shrink our snapshot metadata significantly. However, when the sequence eventually wraps a full metadata edit would be needed to relabel stored exceptions, and so some snapshot creates would take much longer than others. Users generally do not like to wait around while snapshots are created.

## Versioned Pointers

This issue was addressed by introducing a one to one mapping between snapshots and arbitrary version numbers in place of sequence numbers (and in the process notched up yet another example of a problem solved

## Versioned pointers: a new method of representing snapshots

by adding a new layer of indirection). The idea of sequenced pointers thus begat version labels, which begat the notion of "versioned pointers", the subject of this post.

It occurred to me that a pointer version label is similar to a revision control version id, and I proceeded to investigate the question of whether revision control techniques could be applied to volume snapshots. I had experimented with a system where each piece of text is labeled by the version in which it first appears, and by the version in which it disappears (the "birth" and "death" labels, essentially a binary weave). By knowing the hierarchical relationship between versions, the "version tree", it is possible to determine whether any given piece of labeled text belongs to a given version.

Volume versioning differs from text versioning in that data chunks do not appear and disappear, but only change. Each change of chunk data can be viewed as a disappearance followed by an appearance, so only a single label needs to be associated with each change. The other label is implied by the presence of a change labeled by some descendent in the version tree. It became clear that a single version label for each rewrite of a given volume chunk is enough to determine the set of versions to which the written data belongs.

### Version Inheritance

If a given version does not have an exception of its own for a particular logical address, then it inherits an exception from its parent, which in turn inherits from its parent if it does not have its own, and so on up to the root of the version tree. The root implicitly inherits all the chunks of the origin volume. Thus, a snapshot of the origin volume is just a new root of the version tree, with the old root as its child. Similarly, a snapshot of a snapshot is a new child of some version, and inherits all the exception of its parent, exactly as required.

Since we have not so far had an efficient way of creating snapshots of snapshots in ddsnap, the possibly of being able to do so merely by altering a version tree seemed very interesting.

Thus encouraged, I set out to determine whether a stable representation is possible in the sense that no full metadata edit would ever be required to maintain this representation except when a version is deleted. If this were the case then this new technique would be well suited to volume snapshotting, where a large amount of durable metadata has to be maintained and updated with good locality.

### Examples

Here is an example version tree with version nodes labeled in the order created:

## Versioned pointers: a new method of representing snapshots

```
.
`-- C '1003'
  `-- B '1002'
    |-- A '1001'
    `-- D
      |-- E '1005'
      `-- F
        |-- G '1007'
        `-- H '1006'
```

Versions A, B and C are snapshots of the origin. (They appear inverted in the tree because new origin snapshots are added at the root.) All other versions represent snapshots of snapshots. Most of the versions have snapshot tags, in quotes. Those that do not are inaccessible ghost versions, explained below.

Given the exception list `[[A, p1] [B, p2]]` where `p1` and `p2` are physical chunks, and omitting the snapshot tags, we have:

```
.
`-- C
  `-- B => p2
    |-- A => p1
    `-- D => p2
      |-- E => p2
      `-- F => p2
        |-- G => p2
        `-- H => p2
```

Version A is the exclusive owner of chunk `p1` while all other versions except C inherit `p2`. C has no exception at this logical address, therefore inherits a chunk of the origin.

Another way to represent this is to overlay the chunks of the exception list on their respective versions:

```
.
`-- C
  `-- B [p2]
    |-- A [p1]
    `-- D
      |-- E
      `-- F
        |-- G
        `-- H
```

This representation shows both the global version tree and an exception list for a particular logical address.

### Writable Snapshots

## Versioned pointers: a new method of representing snapshots

It is highly desirable that snapshots be writeable. As an example of why one might wish to do this, consider a virtualized server farm where each server has its own, slightly different image of the root volume. For each virtual server instance, a snapshot of a generic root volume is taken and customized for or altered at runtime by one of the server instances. (This also constitutes a use case for snapshots of snapshots, to avoid a clumsy revert of the origin volume for each new server instance.)

Writeable snapshots are problematic in terms of inheritance: if a pointer for a newly allocated snapshot store chunk pointer is labeled with a given version then it may be inherited by other children of the same version, violating the principle of snapshot isolation (a write to a snapshot may not change any other snapshot). My solution is to generate a new, implicit version as a child of the written version, to which the new versioned pointer can be added without affecting any other version.

### Ghost Versions

Adding a new layer of indirection was the method of choice once again: each snapshot is known externally by a numeric "tag" which is the handle for all operations on the snapshot, such as creating a virtual block device for it or deleting it. When a snapshot mapping to a version having one or more children is written for the first time, a new version is generated as above and its tag is reassigned to the new version.

The original version loses its tag and cannot thereafter be accessed externally, becoming a "ghost version". A ghost version exists only to allow its children to continue to inherit any data written to it before it had any children.

Thinking back on it, this was a fairly radical idea because it was far from clear that the implicitly snapshot strategy would not rapidly exhaust the limited supply of version numbers. Remarkably, it turned out to be the case that only a limited number of ghost versions could ever be created, that limit being one less than the number of externally known snapshots. Unfortunately, the proof of this will not fit in the margin of this email.

An essential property of a ghost version is that, because it is not externally accessible and thus cannot be read or written, it need not be a consistent point in time version of a volume. Therefore its exceptions can (and must) be freely cannibalized by other versions as the geometry of the version tree changes and exception lists are modified over time.

To illustrate, given a write to a snapshot with tag '1001' mapping to version A, with one child snapshot:

.

## Versioned pointers: a new method of representing snapshots

```
`-- A '1001'  
  |-- B '1002'
```

A version C is implicitly created to hold the new exception [C, p1], which could not have been added to version A because it would then have been inherited by version B, violating the isolation of snapshot 1002:

```
.  
`-- A  
  |-- B '1002'  
  |-- C [p1] '1001'
```

Version A has become a ghost.

### Version Deletion Problem

When a snapshot with exactly one child is deleted, all the exceptions it owns can simply be relabelled to the child version, and the child version replaces the deleted version in the version tree. This does not violate snapshot isolation because the chunks in question were previously inherited by the child anyway.

When a version with more than one child is deleted, it is not obvious how to relabel exceptions belonging to the deleted version. It would be possible to generate duplicate exceptions sharing the original chunk pointers, but this would raise the specter of metadata that can expand during a delete. When the snapshot store is nearly full the delete could fail for lack of space. Not only would this surprise the user, but in the case of automatic deletion to recover space to service a write to a snapshot, the system could deadlock. There is no such thing as an elegant workaround for an expand-in-delete problem.

### Ghost Versions Redux

The solution I adopted is to leave the version associated with a deleted snapshot in the version tree, along with any inherited exceptions. The version's tag is deleted, creating what I initially called a "zombie" version. Later we discovered that zombie and ghost versions are in fact the same, and in particular, share the property that they cannot proliferate without bound. (This was just one of many close calls for the versioned pointer method, where subtle logic came to the rescue in the face of some seemingly insurmountable problem.)

A ghost version can always be deleted when its child count drops to one at the time one of its remaining two children was deleted. The single remaining child is promoted to take its place in the version tree, and all exceptions belonging to the ghost are relabeled to the child as in the case of explicit deletion of a snapshot with a single child.

This fact is one link in the long chain of reasoning required to show that ghost can never outnumber explicitly created snapshots. As a

## Versioned pointers: a new method of representing snapshots

corollary, each ghost version requires at least two children to force its continued existence. (Before leaping to the conclusion that this shows there are always more explicitly snapshots than ghost versions, consider that both children may themselves be ghosts.)

### Exclusive Exceptions

An exclusive exception is one that is not inherited by any child, either because the owner has no children, or each of its heirs has an exception for the same chunk address, or every version that inherits the exception is a ghost. An exclusive exception's data chunk can always be modified without affecting any other snapshot. In particular, whenever a logical chunk of a given snapshot is written repeatedly with no intervening snapshot creates, all but the first write are guaranteed to be to exclusive exceptions.

### Orphan Exceptions

An orphan exception is an exclusive exception that belongs to a ghost. Because it cannot be read or written directly via a snapshot tag or indirectly by being inherited, it is completely inaccessible and will only waste space in the snapshot store if allowed to exist. Orphan exceptions may be created by writes to snapshots or deletes, and the respective algorithms must detect and delete them immediately to avoid leakage. Potentially long and complex chains of inheritance can be involved, so this requirement gives rise to some particularly subtle rules. For example:

"If a target has no children and no exception then removing it or replacing its ghost parent with no exception by a sibling of the target with no heirs reduces heirs of the parent. If heirs are reduced, a search for a ghost ancestor with an uninherited exception must be performed."

### Reading from a Snapshot

Reading from a snapshot at a given logical chunk address requires determining which exceptions are present in the exception list for that logical address, and are labeled by versions lying between the target version and the root of the snapshot tree. If any such exceptions are found, then the exception furthest from the root owns the snapshot data of interest. Otherwise the data for that logical address resides on the origin volume.

### Writing to a Snapshot

After a write to particular logical chunk of a snapshot an exclusive exception for that chunk of that snapshot will always exist, containing the written data. Before the write, the snapshot may have had no exception but inherited an exception from some other snapshot or inherited a chunk from the origin, or it may have had an exception that

## Versioned pointers: a new method of representing snapshots

is inherited by one or more of its descendants, or it may have already had an exclusive exception. In all cases except the last, an exception is added to the snapshot at that logical address. If an exception is inherited from a ghost and the written snapshot is the sole heir of the exception, then the exception will be relabeled, otherwise a new exception will be created, including allocating a snapshot store chunk to hold the exception data.

### Writing to the Origin

If an origin chunk is inherited by any snapshot, then the data about to be overwritten must be copied to the snapshot store before the new data is written. If the root of the snapshot already has an exception, then the origin chunk is clearly not inherited and nothing needs to be done. Otherwise a new exception for the root version is created and the data to be preserved from the origin is copied to the associated chunk.

### Deleting a Snapshot

Snapshot deletion is by far the most complex aspect of the versioned pointers technique. Various rules must be enforced, for example, that no ghost version may exist with less than two children. If the deleted snapshot has one child then the child will be promoted to be a child of the deleted snapshot's parent. If it has two or more children then the deleted snapshot will become a ghost. If it has no children and its parent is a ghost with two children, then the parent will be deleted and the sibling of the deleted snapshot will be promoted to take the place of the parent.

Besides adjusting the version tree, each exception for the deleted version (or versions if the parent is also deleted) must be either removed from the snapshot metadata if it is not inherited by any version, or relabeled to one of its children if it is. Any exceptions that become orphans as a result of no longer being inherited by the deleted snapshot must be detected and removed.

### Implementation

The attached test program implements all six basic snapshot operations:

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

None of the algorithms are worse than  $O(n)$  where  $n$  is the number of versions in the version tree. Most are  $O(e)$  where  $e$  is the number of exceptions in an exception list. This level of performance was largely achieved through pre-computation of lookup tables to accelerate such

## Versioned pointers: a new method of representing snapshots

operations as determining whether a given version lies on the path from some other version to the root of the version tree. The snapshot delete in particular started as an  $O(n^3)$  algorithm before being gradually improved to  $O(e)$  using mapping tables and a multipass approach.

It is thought that all the algorithms can eventually be improved to  $O(\log n)$  worst case performance. Meanwhile, since  $n$  is limited to a fairly small number by the number of bits available in a version label, real world performance appears entirely satisfactory. Compared to our current representation of snapshot exception data, fitting more data in cache is likely to have a bigger effect than the slightly more expensive base operations.

### Extents

The versioned pointer technique is compatible with extent-based data representations, much more so than the snapshot exception representation it is expected to replace, which ties snapshots together with a sharing bitmap that would be complex to represent in an extent form, as the extents are likely to be different for each snapshot.

On the other hand, each exception represented by a versioned pointer simply needs a count field added to become an extent. Using 64 bit exceptions as we do, there is enough room in the pointer for 48 bits of logical address, addressing an even exabyte of snapshot store, 10 bits of version label allowing 512 user visible snapshots, and a 6 bit extent count, allowing extents up to 256K assuming 4K blocks. Allowing for other overheads, this gives a data to metadata ratio of about 15,000 to one, beyond which there is not a great deal of additional benefit to be obtained from extents.

Extension of the current algorithms to handle extents is in progress.

### Application to filesystems

The versioned pointer technique is by no means limited to representing volume data. It also has promise as a means of implementing filesystem snapshots. Compared to the technique used by WAFL or ZFS, there is no recursive copying of tree-structured metadata. Compared to Btrfs, there are no reference counts. A possible deficiency is the bias towards representing all version information for a given logical address together at the same location in the metadata. If there is a lot of version metadata and only a single snapshot is being accessed, a larger amount of metadata may have to be read. Extending the method to handle extents would mitigate this.

If applied to filesystems, it would be feasible to independently snapshot each file and each directory.

### Fuzzing Test

Versioned pointers: a new method of representing snapshots

## Versioned pointers: a new method of representing snapshots

The attached test program implements a "fuzzing test" to verify empirically the correct implementation of the versioned pointers operations. Without loss of generality, only a single exception list is tested, because each exception list is self contained and independent from all other exception associated with other logical addresses.

Each iteration of the fuzz test randomly selects one of the first five operations and performs it on a random target, then reads every snapshot to verify that all contain the expected data. Various consistency tests are performed, for example:

- \* All exception labels are valid
- \* No deleted version labels an exception
- \* No multiple exceptions with same label in same list
- \* No ghosts have orphan exceptions
- \* No ghost has less than two children
- \* No cycles in the version tree

This fuzzing test has run successfully to ten million iterations. This does not prove that the algorithms are correct, but it is encouraging. Along the way, seemingly endless bugs were discovered especially in the area of ghost exception detection. Some of the invariants described in this note were discovered in response to bugs detected by the fuzzing test, which is not to say that empirical observation is a substitute for logical analysis, but that the underlying logic was most certainly discovered faster than it could have been by logical analysis alone.

### Conclusion

The versioned pointer method is a novel technique for representing versioned disk data, which promises the following benefit to the ddsnap snapshotting block device project:

- 1) Maximum number of snapshots increases
- 2) Metadata shrinks by up to 50%
- 3) Supports instant creation of snapshots of snapshots
- 4) Easier to move to extents for additional compactness

It is also possible that the method may prove useful for filesystem snapshots. A prototype implementation exists, demonstrating the efficacy of the technique and empirically verifying correctness.

### Terminology

**Snapshot:** A volume version tagged with a unique id by which it can be accessed and operated on externally.

**Snapshot tag:** An externally visible 32 bit integer specified by an application programs at the time a snapshot is created. The snapshot

## Versioned pointers: a new method of representing snapshots

tag is used to create a virtual snapshot volume through which the snapshot can be read and written.

**Snapshot chunk:** A chunk in the snapshot store holding data that was either copied from the origin due to a write to an origin logical volume or written to a snapshot logical volume.

**Origin chunk:** A chunk of data on the origin volume that is implicitly shared at the same address by any version that does not have alternate data at that address.

**Version label:** A unique id carried by each node in the version tree.

**Version tree:** The root of the version tree is the most recent snapshot of the origin. Each new version of the origin becomes the new root of the version tree and the parent of the old root. Each new version of a version becomes a child of the existing version.

**Chunk inheritance:** When a new version is created it inherits every chunk of its parent. The root of the version tree inherits every chunk of the origin volume. Care must be taken never to write to an inherited chunk, otherwise all versions inheriting the chunk will be altered in violation of the principle of isolation between versions (the I in ACID).

**Exception list:** A list of exceptions for a given logical chunk address defining which physical chunks belong to which versions. Each exception is said to be "at" that logical address.

**Exception:** A pair [V, p] that appears in an exception list to specify that the physical chunk p is inherited by all nodes in the version subtree rooted at the node labelled V, and bounded by nodes labelled by other exceptions in the same list. Each physical chunk is owned by exactly one exception. An exception labeled by a given version is said to be "at" that version. We may say read or write "to an exception" which really means "to the physical chunk owned by the exception".

**Unique vs shared chunks:** A physical chunk is said to be unique if it is not inherited and shared otherwise. An origin chunk at a given logical address is unique if and only if there is an exception at that address labeled by the root version. A chunk of the origin or a version can be written to without affecting any other version if and only if it is unique.

Regards,

Daniel  
`#include <stdio.h>`  
`#include <inttypes.h>`  
`#include <stdbool.h>`  
`#include <string.h>`

## Versioned pointers: a new method of representing snapshots

```
#include <stdlib.h>
#include <sys/types.h>

#define vecmove(d, s, n) memmove(d, s, (n) * sizeof(*(d)))
#define vecset(d, v, n) memset(d, v, (n) * sizeof(*(d)))
#define error(string, args...) do { printf(string "\n", ##args); exit(99); } while (0)
#define assert(expr) do { if (!(expr)) error("Failed assertion \"%s\"", #expr); } while (0)
#define trace_off(cmd)
#define trace_on(cmd) cmd
#define PACKED

void hexdump(void *data, unsigned size)
{
    while (size) {
        unsigned char *p;
        int w = 16, n = size < w? size: w, pad = w - n;
        printf("%p: ", data);
        for (p = data; p < (unsigned char *)data + n;)
            printf("%02hx ", *p++);
        printf("%*s \\\", pad*3, "");
        for (p = data; p < (unsigned char *)data + n;) {
            int c = *p++;
            printf("%c", c < ' ' || c > 127 ? '!': c);
        }
        printf("\\n");
        data += w;
        size -= n;
    }
}

#define LABEL_BITS 8
#define CHUNK_BITS 54
#define MAXVERSIONS (1 << LABEL_BITS)

typedef uint16_t version_t;
typedef uint16_t label_t;
typedef uint64_t chunk_t;
typedef unsigned tag_t;

struct exception { label_t label: LABEL_BITS; chunk_t chunk: CHUNK_BITS; } PACKED;
struct version { tag_t tag; label_t parent; bool used, ghost, present, pathmap, family; };

struct version ver[MAXVERSIONS];
label_t ordmap[MAXVERSIONS];
label_t children[MAXVERSIONS];
label_t child_count[MAXVERSIONS];
label_t child_index[MAXVERSIONS];
label_t version_count, active_count;
unsigned cycle;

label_t get_child(label_t parent, unsigned i)
```

## Versioned pointers: a new method of representing snapshots

```
{
return children[child_index[parent] + i];
}

label_t get_parent(label_t child)
{
return ver[child].parent;
}

label_t get_root(void)
{
assert(child_count[0]);
return children[0];
}

label_t is_ghost(label_t version)
{
return ver[version].ghost;
}

int find_tag(tag_t tag)
{
for (int version = 1; version < version_count; version++)
if (!is_ghost(version) && ver[version].tag == tag)
return version;
error("invalid snapshot '%u'", tag);
return 0;
}

void show_table(void)
{
for (int i = 0; i < version_count; i++) {
printf("%i: ", i);
if (!ver[i].used)
printf("(free)");
else if (!i)
printf("(origin)");
else {
printf("<- ");
if (!get_parent(i))
printf("root");
else
printf("%i", get_parent(i));
if (!is_ghost(i)) // 0 should be a ghost
printf(" %i", ver[i].tag);
}
printf("\n");
}
}

int count_table(void)
```

## Versioned pointers: a new method of representing snapshots

```
{
int total = 0;
for (int i = 0; i < version_count; i++)
total += ver[i].used;
return total;
}

unsigned exceptions;
struct exception excep[1000];

void show_exceptions(void)
{
printf("%i exceptions: ", exceptions);
for (int i = 0; i < exceptions; i++)
printf("[%i, %Lu] ", excep[i].label, (chunk_t)excep[i].chunk);
printf("\n");
}

void show_index(void)
{
printf("child index: ");
for (int i = 0; i < version_count; i++)
printf("%i:%u ", i, child_index[i]);
printf("\n");
}

int show_subtree(version_t version, int depth, version_t target)
{
assert(depth < MAXVERSIONS);
printf("%*s%i: ", 3 * depth, "", version);
if (!is_ghost(version))
printf("%i", ver[version].tag);
else printf("~%i", child_count[version]);
for (int i = 0; i < exceptions; i++)
if (excep[i].label == version)
printf(" [%Lu]", (chunk_t)excep[i].chunk);
printf("%s\n", target == version ? " <==" : "");
int total = 0;
for (int i = 0; i < child_count[version]; i++)
total += show_subtree(get_child(version, i), depth + 1, target);
return total + 1;
}

void show_tree_with_target(tag_t tag)
{
version_t target = tag == -1 ? 0 : find_tag(tag);
int total = child_count[0] ? show_subtree(get_root(), 0, target) : 0;
printf("(%u versions)\n", total);
}

void show_tree(void)
```

## Versioned pointers: a new method of representing snapshots

```
{
show_tree_with_target(-1);
}

int count_subtree(label_t version, int depth)
{
if (depth > MAXVERSIONS)
return MAXVERSIONS;
assert(ver[version].used);
int total = 0;
for (int i = 0; i < child_count[version]; i++)
total += count_subtree(get_child(version, i), depth + 1);
return total + 1;
}

int count_tree(void)
{
return count_subtree(0, 0);
}

void order_tree(label_t version, int order)
{
ordmap[version] = order;
for (int i = 0; i < child_count[version]; i++)
order_tree(get_child(version, i), order + 1);
}

/* Chunk allocation */

#define MAXCHUNKS MAXVERSIONS

typedef unsigned data_t;

chunk_t nextchunk;
chunk_t checkchunk[MAXVERSIONS];
data_t snapdata[MAXCHUNKS], orgdata = 0x1234;
data_t checkdata[MAXVERSIONS];
bool allocmap[MAXCHUNKS];

chunk_t new_chunk(data_t data)
{
{
for (int i = 0; i < MAXCHUNKS; i++, nextchunk++) {
if (nextchunk == MAXCHUNKS)
nextchunk = 0;
if (!allocmap[nextchunk])
goto found;
}
error("out of chunks");
found:
assert(!allocmap[nextchunk]);
allocmap[nextchunk] = 1;
}
```

## Versioned pointers: a new method of representing snapshots

```
snapdata[nextchunk] = data;
return nextchunk++;

}

void free_chunk(chunk_t chunk)
{
assert(allocmap[chunk]);
allocmap[chunk] = 0;
}

/* Version allocation */

label_t new_version(label_t parent, uint32_t tag)
{
int version;
for (version = 1; version < version_count; version++)
if (!ver[version].used)
goto recycle;
int last = version_count - 1;
child_index[version_count] = version_count ? child_index[last] + child_count[last]: 0;
version = version_count++;
recycle:
ver[version] = (struct version){ .parent = parent, .tag = tag, .used = 1 };
assert(!child_count[version]);
active_count++;
return version;
}

void free_version(label_t version)
{
assert(ver[version].used);
ver[version].parent = 0;
ver[version].used = 0;
active_count--;
}

/* Version tree editing */

void add_exception(label_t label, chunk_t chunk)
{
printf("new exception [%u, %Lu]\n", label, chunk);
assert(exceptions < MAXVERSIONS);
excep[exceptions++] = (struct exception){ .label = label, .chunk = chunk };
}

bool pathmap[MAXVERSIONS][MAXVERSIONS]; // should be sparse vec of bitmaps

/*
* Store the ord numbers in the version table. Per-version bitmap specifies
* whether any given version is on the path to root. Walk the exception list
```

## Versioned pointers: a new method of representing snapshots

```
* looking for the label on the path with the highest ord.
*/
struct exception *lookup_chunk(label_t target)
{
    if (!ver[target].pathmap) {
        trace_off(printf("load pathmap for %u \n", target);)
        memset(pathmap[target], 0, sizeof(pathmap[target]));
        for (label_t v = target; v; v = get_parent(v))
            pathmap[target][v] = true;
        ver[target].pathmap = 1;
    }
    int high = 0;
    bool *path = pathmap[target];
    struct exception *found = NULL;
    for (struct exception *e = excep; e < excep + exceptions; e++)
        if (path[e->label] && ordmap[e->label] > high)
            high = ordmap[(found = e)->label];
    return found;
}
```

```
bool family[MAXVERSIONS][MAXVERSIONS]; // should be sparse vec of bitmaps
```

```
void load_family(label_t target)
{
    trace_off(printf("load family for %u \n", target);)
    memset(family[target], 0, sizeof(family[target]));
    for (int i = 0; i < child_count[target]; i++)
        family[target][get_child(target, i)] = true;
    family[target][target] = true;
    ver[target].family = 1;
}
```

```
unsigned count_family(label_t target)
{
    if (!ver[target].family)
        load_family(target);
    bool *map = family[target];
    int total = 0;
    for (int i = 0; i < exceptions; i++)
        total += map[excep[i].label];
    assert(total <= child_count[target] + 1);
    return total;
}
```

```
struct exception *find_exception(label_t target)
{
    for (int i = 0; i < exceptions; i++)
        if (excep[i].label == target)
            return &excep[i];
    error("label %u missing", target);
}
```

## Versioned pointers: a new method of representing snapshots

```
void set_present(bool flag)
{
for (struct exception *e = excep; e < excep + exceptions; e++)
ver[e->label].present = flag;
}

bool is_present(version_t version)
{
return ver[version].present;
}

int count_heirs(label_t version)
{
int heirs = 0;
for (int i = 0; i < child_count[version]; i++) {
version_t child = get_child(version, i);
if (!is_present(child))
heirs += count_heirs(child) + !is_ghost(child);
}
return heirs;
}

int inherited(version_t version)
{
set_present(1);
int heirs = count_heirs(version);
set_present(0);
//printf("version %i exception has %i heirs\n", version, heirs);
return heirs;
}

label_t *children_p(label_t version)
{
return children + child_index[version];
}

label_t *insert_child_p(label_t parent, label_t child, unsigned count)
{
label_t *p = children_p(parent);
/* insert sorted for cosmetic reasons */
for (int i = 0; i < count; i++, p++)
if (child < *p)
break;
return p;
}

void insert_child(label_t parent, label_t child)
{
label_t *p = insert_child_p(parent, child, child_count[parent]);
vecmove(p + 1, p, children + version_count - p - 1);
}
```

## Versioned pointers: a new method of representing snapshots

```
*p = child;
for (int i = parent + 1; i < version_count; i++)
  child_index[i]++;
child_count[parent]++;
ver[child].parent = parent;
ver[parent].family = 0;
order_tree(get_root(), 1); // overkill
}

label_t *find_child(label_t parent, label_t child)
{
  for (int i = 0; i < child_count[parent]; i++)
    if (get_child(parent, i) == child)
      return children_p(parent) + i;
  error("child not found");
}

void remove_child(label_t child)
{
  label_t parent = get_parent(child);
  label_t *p = find_child(parent, child);
  vecmove(p, p + 1, children + version_count - p - 1);
  for (int i = parent + 1; i < version_count; i++)
    child_index[i]--;
  child_count[parent]--;
  ver[parent].family = 0;
}

void replace_child(label_t child, label_t newchild)
{
  label_t parent = get_parent(child);
  label_t *p1 = children_p(parent), *p2 = find_child(parent, child);
  /* insert sorted for cosmetic reasons */
  vecmove(p2, p2 + 1, p1 + child_count[parent] - p2 - 1);
  p2 = insert_child_p(parent, newchild, child_count[parent] - 1);
  vecmove(p2 + 1, p2, p1 + child_count[parent] - p2 - 1);
  *p2 = newchild;
  ver[newchild].parent = parent;
  free_version(child);
  ver[parent].family = 0;
}

void invalidate_path(version_t version)
{
  assert(version < MAXVERSIONS);
  assert(ver[version].used);
  ver[version].pathmap = 0;
  for (int i = 0; i < child_count[version]; i++)
    invalidate_path(get_child(version, i));
}
```

## Versioned pointers: a new method of representing snapshots

```
void promote_child(label_t child)
{
label_t parent = get_parent(child);
printf("promote version %u to child of %u \n", child, parent);
assert(child_count[parent] == 1);
remove_child(child);
replace_child(parent, child);
invalidate_path(child);
order_tree(get_root(), 1); // overkill
}

void extract_children(void) // O(n^2)
{
unsigned total = 0;

memset(child_count, 0, sizeof child_count);
for (int parent = 0; parent < version_count; parent++) {
child_index[parent] = total;
for (int child = 0; child < version_count; child++)
if (get_parent(child) == parent) {
children[total++] = child;
child_count[parent]++;
}
}
}
/*
* Three pass O(n) extract algorithm
*
* 1: walk the table incrementing child counts of nonfree parents
* 2: accumulate the counts to create the index, clear the counts
* 3: walk the table filling in the children using the index
*/
void extract_children_fast_untested(void) // O(n^2)
{
unsigned total = 0;
memset(child_count, 0, version_count);
for (int i = 0; i < version_count; i++)
if (ver[i].used)
child_count[get_parent(i)]++;
for (int i = 0; i < version_count; i++) {
child_index[i] = total;
total += child_count[i];
}
memset(child_count, 0, version_count);
for (int i = 0; i < version_count; i++)
if (ver[i].used) {
version_t parent = get_parent(i);
children[child_index[parent] + child_count[parent]++] = i;
}
}
}
```

## Versioned pointers: a new method of representing snapshots

```
/*
 * O(n) exception delete
 *
 * 1) walk the exception list incrementing per parent present child counts
 * 2) walk the list deleting target exceptions where present equals child count
 * 3) walk the list clearing present entries for the next time round
 */
label_t brood[MAXVERSIONS]; /* children present in exception list per parent */

bool delete_exceptions(version_t target, version_t parent)
{
    printf("delete target %u, parent %i, %i children\n", target, parent, child_count[target]);
    struct exception *limit = excep + exceptions, *save = excep, *kill = NULL;
    for (struct exception *from = excep; from < limit; from++)
        brood[get_parent(from->label)]++;
    set_present(1);
    if (is_present(target)) {
        if (child_count[target] > 1 && !count_heirs(target))
            kill = find_exception(target);
        } else {
        /* kill orphans */
        version_t ancestor = parent;
        while (!is_present(ancestor) && ancestor)
            ancestor = get_parent(ancestor);
        if (ancestor && is_ghost(ancestor) && is_present(ancestor) && !count_heirs(ancestor))
            kill = find_exception(ancestor);
        }
    if (kill)
        printf("kill ancestor %u with %u heirs\n", kill->label, count_heirs(kill->label));
    if (!is_ghost(parent))
        parent = 0;
    for (struct exception *from = excep; from < limit; from++) {
        version_t label = from->label;
        if (kill == from)
            goto free;
        if (label == target || label == parent) {
            if (child_count[label] == brood[label])
                goto free;
            if (child_count[label] == 1) {
                if (!count_heirs(label))
                    goto free;
                printf("relabel %i as %i\n", label, get_child(label, 0));
                ver[label].present = 0;
                label = from->label = get_child(label, 0);
                ver[label].present = 1;
                goto keep;
            }
        }
        goto keep;
    }
keep:
}
```

## Versioned pointers: a new method of representing snapshots

```
*save++ = *from;
continue;
free:
ver[label].present = 0;
printf("free [%i, %Li]\n", from->label, (chunk_t)from->chunk);
free_chunk(from->chunk);
exceptions--;
}
set_present(0);
for (save = excep; save < excep + exceptions; save++)
brood[get_parent(save->label)] = 0;
return parent && child_count[parent] == 1;
}

/* External operations */

void delete_snapshot(tag_t tag)
{
//if (cycle == 75109) show_tree_with_target(tag);
version_t target = find_tag(tag);
memset(brood, 0, sizeof(brood));
ver[target].tag = 0;
ver[target].ghost = 1; /* does not inherit ghost exception */
version_t parent = get_parent(target);
switch (child_count[target]) {
case 0:;
remove_child(target); /* no relabel to deleted child */
free_version(target);
if (delete_exceptions(target, parent))
promote_child(get_child(parent, 0));
break;
case 1:
delete_exceptions(target, parent);
promote_child(get_child(target, 0));
break;
default:
delete_exceptions(target, parent);
}
}
/*
* Ghost exception inheritance
*
* Any ghost exception inherited only by ghosts may be deleted.
*
* If a target with more than one child, an exception and no heirs is deleted
* then the exception may be deleted.
*
* Replacing a target with one child and no exception by its child with no
* heirs reduces heirs of the parent.
*
* If a target has no children and no exception then removing it or replacing
```

## Versioned pointers: a new method of representing snapshots

```
* its ghost parent with no exception by a sibling of the target with no
* heirs reduces heirs of the parent.
*
* If heirs are reduced, a search for a ghost ancestor with an uninherited
* exception must be performed.
*/
```

```
void snapshot_of_snapshot(tag_t tag, tag_t parent_tag)
{
    label_t parent = find_tag(parent_tag);
    label_t child = new_version(parent, tag);
    assert(!child_count[child]);
    insert_child(parent, child);
    order_tree(get_root(), 1); // overkill
}
```

```
void snapshot_of_origin(tag_t tag)
{
    label_t root = new_version(0, tag);
    if (!child_count[0]) {
        insert_child(0, root);
        return;
    }
    insert_child(root, get_child(0, 0));
    children[0] = root;
    invalidate_path(root);
    order_tree(get_root(), 1); // overkill
}
```

```
data_t read_snapshot(tag_t tag)
{
    struct exception *found = lookup_chunk(find_tag(tag));
    //printf("read version %u, chunk %Li \n", find_tag(tag), found->chunk);
    return found ? snapdata[found->chunk] : orgdata;
}
```

```
void write_snapshot(tag_t tag, data_t data)
{
    label_t target = find_tag(tag);
    printf("write 0x%x to snapshot %i version %u \n", data, tag, target);
    struct exception *e;

    if (count_family(target) == child_count[target] + 1) {
        e = find_exception(target);
        goto rewrite;
    }
```

```
    if (child_count[target]) {
        label_t child = new_version(target, tag);
        printf("implicit version %u of %u \n", child, target);
        insert_child(target, child);
    }
```

## Versioned pointers: a new method of representing snapshots

```
ver[target].ghost = 1;
target = child;
}

set_present(1);
label_t ancestor = get_parent(target);
while (!is_present(ancestor) && is_ghost(ancestor))
    ancestor = get_parent(ancestor);
bool relabel = is_ghost(ancestor) && count_heirs(ancestor) == 1;
set_present(0);

if (relabel) {
    printf("relabel version %u exception to %u!\n", ancestor, target);
    e = find_exception(ancestor);
    e->label = target;
    goto rewrite;
}

chunk_t chunk = new_chunk(data);
add_exception(target, chunk);
checkchunk[target] = chunk;
checkdata[target] = data;
snapdata[chunk] = data;
return;

rewrite:
printf("rewrite chunk %Lu to 0x%x\n", (chunk_t)e->chunk, data);
checkdata[target] = data;
snapdata[e->chunk] = data;
return;
}

/*
 * O(n) search for one child not present
 *
 * 1) walk the exception list setting each child present
 * 2) walk the child list to find the one not present
 * 3) walk the exception list clearing present for next time round
 */

void write_origin(data_t data)
{
    printf("write 0x%x to origin\n", data);
    if (cycle == 10901)
        show_tree();
    if (!child_count[0])
        goto write;
    label_t version = get_root();
    for (int i = 0; i < exceptions; i++)
        if (except[i].label == version)
            goto write;
}
```

## Versioned pointers: a new method of representing snapshots

```
if (1 && is_ghost(version)) { // !!! this code is probably bogus
/* do not add unique exception to ghost */
set_present(1);
while (count_family(version) == child_count[version] - 1) {
int i;
for (i = 0; i < child_count[version]; i++)
if (!ver[get_child(version, i)].present)
goto deeper;
error("did not find only child not present");
deeper:
version = get_child(version, i);
printf("deep relabel to %u, children %u\n", version, child_count[version]);
if (!is_ghost(version))
break;
}
bool clobber = is_ghost(version) && !count_heirs(version);
set_present(0);
if (clobber)
goto write;
}
add_exception(version, new_chunk(orgdata));
write:
orgdata = data;
}

void fuzztest(int cycles)
{
tag_t snap[MAXVERSIONS], tag, newtag = 1000;
int snaps = 0;
char *why;

for (cycle = 1; cycle <= cycles; cycle++) {
printf("--- cycle %i ---\n", cycle);
if (!snaps || rand() % 5 == 0) {
if (!snaps || (snaps < MAXVERSIONS / 2 && rand() % 2000000 < 1000000)) {
/* Randomly create snapshot */
tag = snap[snaps] = newtag++;
if (!snaps || rand() % 20 == 0) {
printf("create snapshot %u of origin\n", tag);
snapshot_of_origin(tag);
checkdata[find_tag(tag)] = orgdata;
} else {
tag_t parent = snap[rand() % snaps];
printf("create snapshot %u of %u\n", tag, parent);
snapshot_of_snapshot(tag, parent);
checkdata[find_tag(tag)] = read_snapshot(parent);
}
snaps++;
} else {
/* Randomly delete snapshot */
int which = rand() % snaps;
```

## Versioned pointers: a new method of representing snapshots

```
printf("delete snapshot %u \n", snap[which]);
delete_snapshot(snap[which]);
why = "delete left wrong number of versions in version tree";
if (count_tree() != active_count)
goto eek;
snap[which] = snap[--snaps];
}
} else {
/* Write to random snapshot */
data_t data = rand();
if (rand() % 20 == 0) {
tag = -1;
write_origin(data);
} else {
tag = snap[rand() % snaps];
write_snapshot(tag, data);
}
}
continue;
why = "version 0 corrupted";
if (is_ghost(0) || child_count[0] > 1)
goto eek;
why = "write left wrong number of versions in version tree";
if (count_tree() != active_count)
goto eek;
/* Verify valid exception list */
bool member[MAXVERSIONS] = { };
for (int i = 0; i < exceptions; i++) {
label_t version = excep[i].label;
//printf("[%i, %Lu]\n", version, (chunk_t)excep[i].chunk);
why = "invalid exception label";
if (version == 0 || version > MAXVERSIONS)
goto eek;
why = "deleted version in exception list";
if (!ver[version].used)
goto eek;
why = "multiple exceptions with same label";
if (member[version])
goto eek;
why = "ghost has inaccessible exception";
if (is_ghost(version) && !inherited(version)) {
printf("ghost %i has inaccessible exception\n", version);
goto eek;
}
member[version] = 1;
}
for (int version = 0; version < version_count; version++) {
why = "present flag should be clear";
if (ver[version].present) {
printf("present flag should be clear for %u\n", version);
goto eek;
}
```

## Versioned pointers: a new method of representing snapshots

```
}
why = "ghost has less than two children";
if (is_ghost(version) && child_count[version] < 2) {
printf("ghost %i has less than two children\n", version);
goto eek;
}
}
why = "tree has a cycle";
int counted = count_tree();
if (counted == MAXVERSIONS)
goto eek;
why = "wrong number of versions in version tree";
if (counted != active_count)
goto eek;
why = "snapshot has wrong data after write";
for (int i = 0; i < snaps; i++) {
data_t data = read_snapshot(snap[i]);
if (data != checkdata[find_tag(snap[i])]) {
printf("snapshot %u has wrong data 0x%x\n", snap[i], data);
tag = snap[i];
goto eek;
}
}
//if (cycle == 75109) { show_tree(); exit(1); }
}
show_tree();
show_exceptions();
return;
eek:
printf("---- Failed at cycle %u ---- \n", cycle);
show_tree_with_target(tag);
//show_table();
printf("tree count = %u, table count = %u, active count = %u\n", count_tree(), count_table(), active_count);
show_exceptions();
error("%s", why);
}

int main(void)
{
label_t v0 = new_version(-1, 0);

#if 1
srand(123);
fuzztest(1000000);
return 0;
#endif

tag_t nexttag = 1001;
label_t v1 = v1 = new_version(v0, nexttag++);
label_t v2 = v2 = new_version(v1, nexttag++);
label_t v3 = v3 = new_version(v2, nexttag++);
```

## Versioned pointers: a new method of representing snapshots

```
#if 0
show_table();
extract_children();
show_tree();
hexdump(child_count, 16);
hexdump(child_index, 16);
hexdump(children, 16);
promote_child(v3);
//remove_version(v5);
//delete_snapshot(2000);
//nested_snapshot(123, 2005);
//snapshot_of_origin(123);
show_table();
hexdump(child_count, 16);
hexdump(child_index, 16);
hexdump(children, 16);
show_tree();
extract_children();
show_tree();
return 0;
free_version(v1);
free_version(v4);
show_table();
return 0;
#endif
extract_children();
label_t target = v2;
tag_t tag = ver[target].tag;
add_exception(v2, new_chunk(0));
show_exceptions();
show_tree();
printf("data = %u\n", read_snapshot(tag));
write_snapshot(tag, 0x333);
show_tree();
write_snapshot(1003, 0x666);
show_tree();
// hexdump(snapdata, 16);
// write_origin(666);
// write_origin(777);
// write_snapshot(ver[target].tag, 555);
show_exceptions();
hexdump(snapdata, 32);
printf("data = 0x%x, orgdata = 0x%x\n", read_snapshot(tag), orgdata);
printf("v3 data = 0x%x\n", read_snapshot(ver[v3].tag));
show_tree();
hexdump(family[v2], 16);
// delete_exceptions((label_t[]){ v7 }, 1);
delete_snapshot(1003);
show_tree();
show_exceptions();
delete_snapshot(1001);
```

## Versioned pointers: a new method of representing snapshots

```
show_tree();
show_exceptions();
delete_snapshot(1002);
show_tree();
show_exceptions();
snapshot_of_origin(1009);
show_tree();
show_exceptions();
printf("data = 0x%x, orgdata = 0x%x\n", read_snapshot(1002), orgdata);
hexdump(child_index, 16);
return 0;

label_t v4 = v4 = new_version(v1, nexttag++);
label_t v5 = v5 = new_version(v4, nexttag++);
label_t v6 = v6 = new_version(v4, nexttag++);
add_exception(v5, new_chunk(0));
add_exception(v6, new_chunk(0));
#if 0
load_family(v4);
hexdump(family[v4], 16);
return 0;
#endif

return 0;
}
```