Snapper (v2) - A PhantomOS Snapshot Mechanism

Rumen Mitov

2025-06-04

Contents

1 Introduction

Snapper 2.0 (henceforth known as just Snapper) is a snapshot mechanism for capturing the state of a set of components (e.g. virtual memory pages). It uses the a logging file system (e.g. ext4) to ensure that file operations (e.g. modification or deletion) are resistant to OS crashes mid-way through the operation.

A vital property of Snapper is that it needs to be disk space efficient in order not to bloat the file system. This is done by utilizing a mapping from the component identifier to a file containing the component's state at the time of the snapshot. Thus if a component's state remains unchanged the mapping will point to a file from a previous snapshot, hence no new file needs to be created.

2 The Problems Snapper Solves

Snapper is trying to build upon previous attempts to create a snapshot mechanism for PhantomOS. Snapper is specifically designed for it to be able to be implemented in Genode[1]. Here are the problems faced by previous snapshot mechanisms and how Snapper solves them:

2.1 Disk-Space Efficiency for Multiple Snapshots

Snapper only stores new data in its snapshots. If a page has not changed, its state will be recovered from previous snapshots. This reduces disk usage when compared with a "superblock" mechanism, where the entire snapshot data is bundled in one large object.

2.2 Handling of Snapshot Files

A previous idea to hold on to old, but still in-use, snapshot files was to use hardlinks. A hardlink would be created from the old snapshot directory to the current snapshot directory if a file had not changed.

Unfortunately, Genode does not provide support for hardlinks[2] and I was unsuccessful in patching hardlinks in. Snapper solves this by via a table that maps the virtual pages to the path of the file currently containing their content. Additionally, a reference count will be kept for each file. This tracks

[1] `https://genode.org/`

[2] See discussion: `https://lists.genode.org/mailman3/hyperkitty/list/users@lists.genode.org/thread/TKLOW3SZLHVOGW453TM5G2AQTXQWEMLF/`

how many generations (i.e. snapshots) require this file. Once the reference count reaches 0, the file will be deleted.

2.3 File Integrity

Snapper should be resilient towards filesystem failures (such as bad unmount) as it utilizes ext4 and thus utilities like fsck can be used to recover the filesystem state.

In addition to this, Snapper provides integrity checks for individual files in the form of a **hash** representing the data stored in the file. This is used to track whether or not the saved virtual page contents have been modified before being recovered. See the Configuration section on how to control the policy for failed integrity checks.

Another use case of the hash is that it provides a way to know when a new file needs to be created for a snapshot (i.e. when the virtual page has changed since the last snapshot).

2.4 Redundancy

The more redundancy there is between snapshots, the more robust the snapshot mechanism is. In order to achieve said redundancy, once a backing file has too many snapshots that depend on it (i.e. its reference count is greater or equal to `Snapper::Config::redundancy`) Snapper will create a copy of the file and map all future snapshots with the copy and the original.

By configuring `Snapper::Config::redundancy`, the system administrator can control how often redundant file copies are made, leaving them to decide the right balance between robustness and storage usage.

3 Comparison of Snapper 2.0 and Prior Snapshot Mechanisms

Feature	Superblock	Squid	Snapper 1.0	Snapper 2.0
Redundancy	x			x
Multiple Generations		x		x
Integrity Checks	x			x
File Cleanup	x	x		x
Genode Compatibility	x		x	x
Makes Use of Ext4		x	x	x

3.1 Snapper 2.0 vs Superblock

Feature	Superblock	Snapper 2.0
Redundancy	makes two copies of the data and stores them in two regions	makes copies of files that are used in many generations
Multiple Generations	n/a	links identical files from a previous snapshot
Integrity Checks	checksum	cyclic redundancy checks
File Cleanup	old superblocks are removed	files with a reference count of 0 are removed
Genode Compatibility	compatible	compatible
Makes Use of Ext4	n/a	uses ext4's journaling capabilities

While Superblock has a fair amount of redundancy, a system administrator does not have the same level of control as with Snapper 2.0. Once a superblock is created a copy of it is saved in another location on disk. In contrast, Snapper 2.0 supports multiple copies of the same file. The system administrator can decide how often these copies are made by setting the constant `Snapper::Config::redundancy`.

A limitation of the Superblock implementation is that it has no support for multiple generations of snapshots. The mechanism only keeps a superblock of the current system state along with a redundant copy of it. While this serves the primary use case for a snapshot (that being restoring the system state after a crash), it lacks the flexibility of Snapper 2.0 when it comes to multiple versions of the system.

3.2 Snapper 2.0 vs Squid

Feature	Squid	Snapper 2.0
Redundancy	n/a	makes copies of files that are used in many generations
Multiple Generations	links identical files from a previous snapshot	links identical files from a previous snapshot
Integrity Checks	n/a	cyclic redundancy checks
File Cleanup	when the last link of a file is gone, it is removed	files with a reference count of 0 are removed
Genode Compatibility	incompatible (Genode does not support hardlinks)	compatible
Makes Use of Ext4	uses ext4's journaling capabilities	uses ext4's journaling capabilities

The Squid Snapshot mechanism sought to improve on the Superblock mechanism by saving each snapshot in its own directory and using hardlinks for the virtual pages whose contents had not changed since the previous snapshot. This would solve the issue of unnecessary duplication of data while also providing the functionality of multiple generations of snapshots.

I could not get this approach to work, however, as Genode does not support hardlinks and, after failing to add them to the virtual filesystem, I decided to give up on this approach.

Snapper 2.0 is most similar to Squid in terms of the underlying mechanism. Snapper 2.0 uses a mapping from virtual page number to file path to keep track of where the data is stored and to avoid duplication. Additionally, each file keeps track of how many generations it appears in, and when that number reaches 0 Snapper 2.0 knows that this file can be removed. This is done to replicate hardlink functionality without actually implementing hardlinks in the virtual filesystem.

3.3 Snapper 2.0 vs Snapper 1.0

Feature	Snapper 1.0	Snapper 2.0
Redundancy	n/a	makes copies of files that appear in many generations
Multiple Generations	n/a (only latest snapshot can be recovered from)	links identical files from a previous snapshot
Integrity Checks	n/a	cyclic redundancy checks
File Cleanup	possibility of a leak in disk storage	files with a reference count of 0 are removed
Genode Compatibility	compatible	compatible
Makes Use of Ext4	uses ext4's journaling capabilities	uses ext4's journaling capabilities

The first version of Snapper kept track of which virtual page was backed by which file via mappings in the singleton `SnapTable` and a mapping from file to virtual page managed by the singleton `ReverseTable`. Snapper 1.0 required both tables so that files in-use can be identified and all other unnecessary files could be removed.

The shortcoming of Snapper 1.0 was that it used singletons to manage the mappings. This meant that only one version of the system state could be had at a given time. Consequently, if the file that stored `SnapTable` and `ReverseTable` were to be corrupted, not only would there be no consistent state which the system could recover, but information about which file was in-use would be lost, leading to "zombie" files which the mechanism would never delete as it would have lost information on their existence.

Another (minor) issue with Snapper 1.0 was that Genode's Dictionary implementation is unsuitable for the use cases of the mechanism (e.g. no support for iterating over entries) and thus a Dictionary would need to be implemented which adds more complexity to the mechanism. Snapper 2.0, on the other hand, uses arrays for the mappings to avoid this complexity.

As for the major pitfalls of Snapper 1.0, Snapper 2.0 uses an archive file for each snapshot generation. This archive file contains the mapping for the current generation, meaning that any generation could be recovered if it has a valid archive file. Moreover, the hash of the archive file is saved alongside the data to ensure that any modifications are detected and the system can react as dictated by the policy. Unlike its predecessor, Snapper 2.0 supports an arbitrary number of prior snapshot generations and it provides integrity checks for all files.

4 Definitions and Notations

4.1 Snapper

Snapper is the name of the snapshot mechanism.

4.2 Component

A discretionary object with a state. The set of all components will henceforth be denoted by P.

4.3 Backlink

A file path leading to a file that contains the data for a component in a given generation. A component may have multiple backlinks in a generation for redundancy. The set of all backlinks for a page in a given snapshot will be denoted by B.

4.4 Snapshot

Structure that contains partial or complete information about the states of the components at a particular point in time.

4.5 Generation

A generation is a completed snapshot, meaning it can restore the component space P.

4.6 Zombie

A file with a reference count greater than one, which is not needed in any generation.

4.7 Dead Snapshot

An invalid generation that contains backlinks, needed for other generations, and possibly zombie files.

4.8 Snapshot Files

The set of all files that contain data on the pages from different snapshots will be denoted by H.

4.9 Archiver

The mechanism that maps $P \rightarrow B$, if a component p_i has its current or past contents saved in a file $h_i \in B$.

4.10 Snapper Root

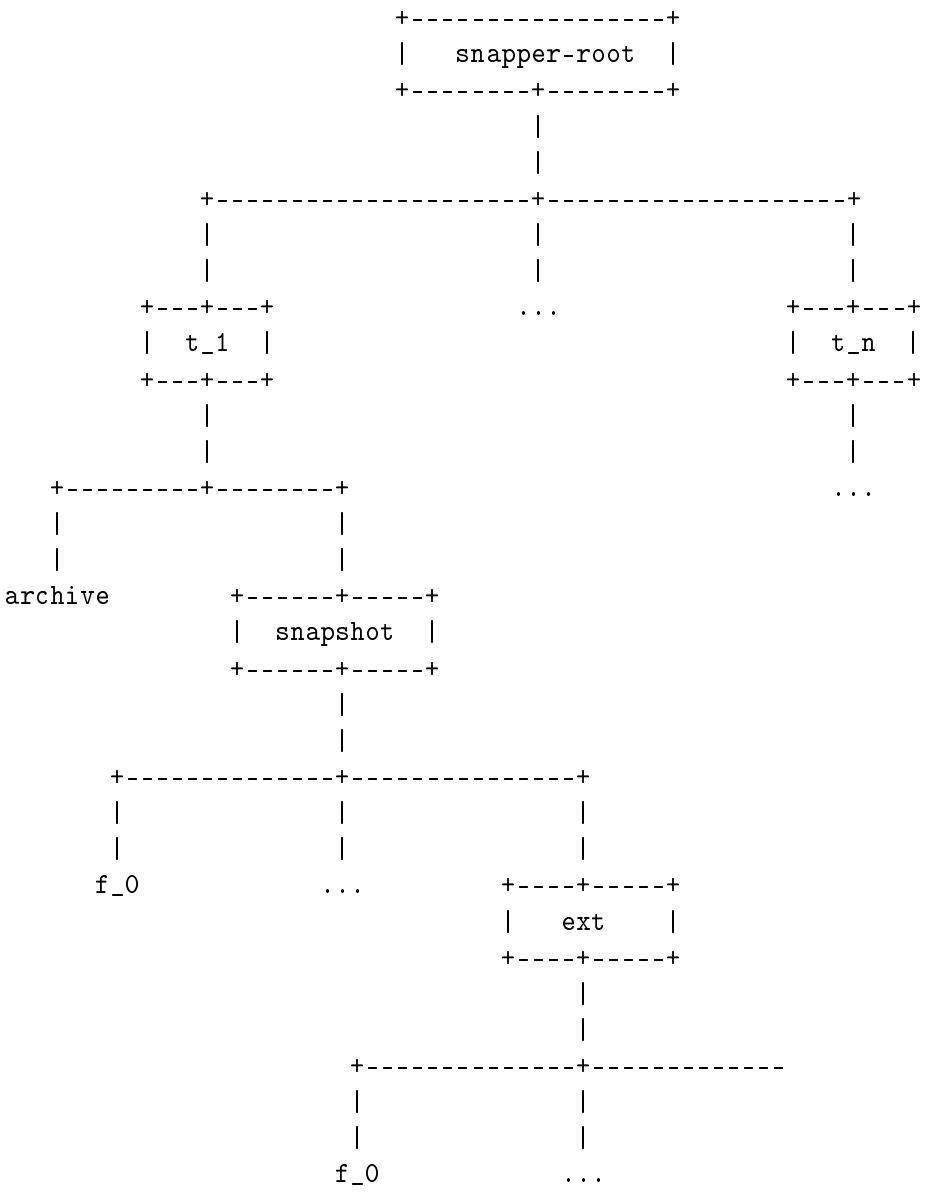
Denoted by <snapper-root> it contains the directories which hold the information for the various snapshots.

4.11 Snapshot Root

A directory containing H' $\subseteq$ H, where H' is the set of all snapshot files that were created for the current snapshot (i.e. indicating that a page's value has changed since the last snapshot).

5 The Snapper Components

5.1 The Snapper Root



- t_i := RTC timestamp of when the i-th generation was finalized
- archive := file that stores the mapping from a page to a file storing its contents for this snapshot
- f_i := snapshot files, named through an incrementing counter (in hex) which is reset for each new sub-level in the hierarchy
- ext := extender directory contains the next level of files

5.2 The Archive File

The archive file contains keeps track of which file is storing the contents of a given component in the current generation. The archive file is a key component of a generation. Without it, a generation is **invalid / dead** (i.e. the system cannot recover the state of the generation). Note, that a generation can be invalid but still be needed for Snapper, as other generations might have a need of files contained within it.

The mapping itself is stored as a Genode dictionary, with the key corresponding to the page number and value contents being a Genode FIFO queue which stores the backlink file paths (relative to <snapper-root>).

For example:

```
Snapper::Archiver[i] = [ "/t_1/snapshot/ext/ext/00cd" ]
```

stores the contents of component p_i in a file found in the generation t_1. Notice how the file path is relative to the <snapper-root>.

Note that the mapping can include multiple backlinks, each of which is a redundant copy of the component's data. If one backlink is missing or has an invalid hash, Snapper will try to recover the next backlink until it either succeeds or it runs out of backlinks.

An example of a mapping entry with multiple backlinks:

```
Snapper::Archiver[i] = [ "/t_1/snapshot/ext/ext/00cd", "/t_0/snapshot/ext/ext/0054" ]
```

where the first file (base name of "00cd") is an identical copy of the second file (base name of "0054") and they are store the contents of component p_i.

The above examples showcase how the backlinks are stored in memory during the lifetime of the `Snapper` object. When it comes to storing the backlinks in the actual archive file, the FIFO queue is expanded such that each backlink is written to the archive file as an individual mapping. Then, when Snapper reads the archive file, it aggregates all mappings with identical

keys into a FIFO queue and that is what constitutes a `Snapper::Archiver` entry.

Example[3] of an archive file's data section corresponding to the multi-backlink example from above:

```
|  KEY  |            VALUE             |
|-------+------------------------------|
|   i   | "/t_1/snapshot/ext/ext/00cd" |
|   i   | "/t_0/snapshot/ext/ext/0054" |
|  ...  |             ...              |
| i + j |             ...              |
```

The archive file has the following structure:

```
+----+----------------+---------------------------------+---------...
| v  | hash           | n                               | data
+----+----------------+---------------------------------+---------...
```

Symbol	Size	Description
v	1 byte	Snapper version
hash	4 bytes	cyclic redundancy check for the data
n	8 bytes	number of entries in the data
data	as required	array that contains snapshot files' paths

The archive file contains information of how many entries comprise the data in order to prevent a `while(true)` loop when reading the data. Note, that the hash applies only to the data which is sufficient as modifying the n (i.e. the number of entries) and appending false entries to the data will result in the overall data segment having a different hash than the original.

Also note, that if entries are appended to the data outside of the snapshot mechanism (i.e. from a malicious third party), when reading the archive file, the snapshot mechanism will read the data up to n entries. All other entries will be disregarded.

[3] In Snapper's implementation the archive file will contain binary data. The example uses plaintext for demonstration purposes.

5.3 The Snapshot File

The snapshot file primarily stores the binary data of an arbitrary page from a given snapshot. Additionally, a snapshot file has a reference counter. The file will be deleted if the reference count were to reach 0. The file also contains a hash which is used for integrity checking and for comparison operations.

The structure of the snapshot file is as follows:

```
+----+----------------+----+---------...
| v  | hash           | rc | data
+----+----------------+----+---------...
```

Symbol	Size	Description
v	1 byte	Snapper version
hash	4 bytes	cyclic redundancy check for the data
rc	1 byte	Reference count (unsigned)
data	as required	the saved page contents

5.4 The Extender Directory

The extender directory is used to reduce the load on the filesystem. Since performance can be impacted if too many files are in the same directory, after a certain number (`Snapper::Config::threshold`), a sub-directory will be created called ext and subsequent snapshot files will be stored within it, instead of the current one. Important to note is that the incremental counter used to name the snapshot files resets within the extender directory.

5.5 The Snapshot Directory

This directory is organized as a radix trie containing all snapshot files of components that have changed since the last generation. Files are added in the extender directories. The extender directories are removed if their last entity (file or sub-directory) gets removed.

5.6 The Generation Directory

The generation directory contains the archive file and the snapshot directory. The directory is uses an RTC timestamp as its name, which is generated at the time of the directory's creation.

The generation directory makes up a complete snapshot. As long as the archive file is present and its hash is valid, the generation should be able to be recovered.

The generation directory is removed when both the snapshot directory and the archive file have been removed.

6 The Snapper Mechanism

6.1 Snapshot Step

The rationale behind this step is to use a file (the archive file) to keep track of the snapshot file(s) (a.k.a backlinks) of the components. This allows for a single source of truth. If the archive file is corrupted it must be replaced with a backup version of the file.

Each snapshot file must keep a reference count which keep track of the number of generations that need this file. Should the reference count exceed `Snapper::Config::redundancy` a new snapshot file will be created to store the data. Both the new file and the old file will be stored as backlinks for later snapshots.

The snapshot process has been designed to allow the snapshot of individual components to be done at an arbitrary time. For example, the user can snapshot the first n-components, then do some computations, and then snapshot the rest of the components. This is not recommended as the state could have changed for the first n-components before the rest are saved in the snapshot, hence leading to an inconsistent system state. It is up to the user to determine if it is more desirable to "pause" the snapshot process, or do it all in one go.

In order to support this flexibility, the Snapper initiate the snapshot procedure. Once this procedure is active the only Snapper operation allowed is the taking of snapshots (i.e. recovering and purging are disallowed). Once all components have been captured in the snapshot, the generation will be committed and Snapper will be returned to its dormant state.

In terms of performance, the taking of snapshots is cheap as it comprises a dictionary lookup, and in the worst case (the page's contents do not appear in a prior generation): a write to a file. The true cost comes when commit-

ting the generation. The `Snapper::Archiver` is written to the generation's archive file, which means iterating over all the entries.

As an optimizations, reference counts are updated during the taking of the snapshot (as opposed to the commit step) in order to prevent opening all the files again at the commit stage. Unfortunately, this could lead to zombie files if the system crashes before the generation is committed (and the archive file is written). If no archive file has any mention of a snapshot file, then that file will never be deleted as its reference count falsely indicates that the file is still being needed. To clean up zombie file please refer to Purge Step (Zombies).

1. If the latest generation does not have a valid archive file, delete it (the generation is incomplete).
2. Initialize a new generation directory with an RTC timestamp as the name.
3. Within the generation directory create the archive file and the snapshot directory.
4. Check if there is a valid prior generation (based on the timestamps). If there is, load the archive file's data into the `Snapper::Archiver`.
5. Let h_i := `Snapper::Archiver[i]`. If `Snapper::Archiver[i]` contains backlinks, use the *first backlink* (i.e. the earliest backlink).
6. For each $p_i \in P$ where the hash of the file h_i does not match the hash of p_i (or h_i does not exist):
   (a) Create new file, h_j, and save the binary contents of p_i into this new file.
   (b) Initialize the snapshot file with the new hash of the data, a reference count of 1, and the binary data of p_i.
   (c) Update `Snapper::Archiver[i]` $\leftarrow$ *path(* h_j *)*, there *path()* is the path relative to <snapper-root>.
7. For each $p_i \in P$ where hash of the file h_i matches the hash of p_i:
   (a) If the file h_i has a reference count greater than or equal to `Snapper::Config::redundancy`:
      i. Create a new file h_j as outlined in Step 6.
      ii. Increment the reference count for all files in `Snapper::Archiver[i]`.
      iii. Enqueue *path(* h_j *)* to `Snapper::Archiver[i]`.

(b) If the file h_i has a reference count lower than `Snapper::Config::redundancy`, increment the reference count of it and all other redundant files in `Snapper::Archiver[i]`.

8. Save `Snapper::Archiver` into the archive file and calculate the hash of the entries.

6.2 Recovery Step

This step uses the archive file to efficiently lookup the data belonging to a page. The recovery process is flexible enough to allow partial recovery, i.e. the user recovers only the pages that they need. The pages can be recovered at any time while the recovery procedure is active. Throughout the recovery process all other Snapper procedures are disallowed.

Entry lookups happen in logarithmic time due to Genode's Dictionary use of AVL-trees. Additionally, in the case that an archive entry's backlinks are invalid a linear search through a queue is used until a valid backlink is found or the queue is exhausted.

A downside to the lookup table being loaded in memory is that more information (i.e. entries and backlinks) result in heavier RAM usage.

1. Choose a generation to boot from (by default the latest one).
2. Check if the generation is valid (i.e. has an archive file with a valid hash). If not, recovery is not possible.
3. Load the archive file of the latest valid generation into `Snapper::Archiver`.
4. For each h $\in$ `Snapper::Archiver` and for each backlink, $h_i \in h$:
   (a) Check the hash with the stored data.
   (b) If h_i does not exist or there is a mismatch with the hash, try the next backlink.
   (c) If there are no more backlinks to check, respond according to the configured policy.
   (d) If the hash matches h_i, load the data of h_i into the corresponding page p_i.

6.3 Purge Step

To purge a generation, the archive file is loaded into memory and each backlink's reference count is decremented. When a file's reference count is decremented to 0, the file is removed. If a directory becomes empty as a result, it is removed. This ensures that all files needed by other generations are kept in the same place, and everything else is properly cleaned up.

The worst case of this approach would be that a snapshot file could be many sub-directories deep and while it is needed all those sub-directories will remain. This cost is minimal and necessary as the alternative would be to

move the file higher in the directory structure, then search for all references to that file in all of the other archive files and update the path, a much more costly endeavor.

1. Make sure the generation is valid (i.e. it has an archive file with a valid hash).

2. If the archive file has an invalid hash:

   (a) If `Snapper::Config::integrity` is set to true, crash the system and ask the system administrator to replace the generation's corrupted archive file with a backup copy.

   Note, that if no backup copy exists it is highly recommended to manually remove the current generation as well as all subsequent generations. Snapper can continue to function without the removal, but the broken generation and its files will never be removed. Alternatively, the administrator could manually remove the broken generation and set `Snapper::Config::integrity` to false. That way any snapshots that relied on the broken generation will only output warnings but will not crash the system if they are unable to recover a file.

   (b) Otherwise, log an error message and boot the system into a clean state.

3. If the archive file has a valid hash:

   (a) Load the archive file into `Snapper::Archiver`.

   (b) For each entry $h \in$ `Snapper::Archiver` and for each file $h_i \in h$: decrement the file h_i's reference count.

   (c) Delete the archive file.

6.4 Purge Step (Zombies)

There is a possibility of files which are no longer in need by any generation (aka zombie files) to occur as a result of a system crash during the Snapshot Step or the Purge Step when the reference counts of files are updated. The system crash would create an inconsistency between the file reference count and the amount of generations that the file is needed by. By design, this inconsistency would always result in the file reference count being lower than the actual file references. Thus, when all the generations referencing

the file have been purged, the file itself will not be purged as its reference count incorrectly states that it is still in need.

To remove **all** zombie files from the system we use the following algorithm. Note that this algorithm is very slow (especially for large component sets and many snapshots) hence it should be used rarely. To achieve its effect run it **at most** once per Purge Step (the only time when zombies may appear).

1. For each **dead snapshot**:
   (a) For each file:
      i. Check if the file is needed in any of the generations (requires linear search).
      ii. If the file is needed by at least one, keep it.
      iii. Otherwise delete it.

7 Snapper's Time and Space Complexity

The following complexity analysis uses the following assumptions:

- let P be the set of all components, and let p = |P|
- let H be the set of all entries in the mapping stored in an archive file, and let h = |H|
- let B be the set of all backlinks present in an archive file, and let b = |B|
- let S be the set of all generations, and let s = |S|
- let Z be the set of all files in dead snapshots, and let z = |Z|

  NOTE: Since Genode's Dictionary uses an AVL-tree, all Dictionary lookups are O(log(n)).

Use-Case	Time Complexity	Space Complexity	Regularity

Continued on next page

Continued from previous page

Use-Case	Time Complexity	Space Complexity	Regularity
Begin snapshot procedure.	O(1)	O(1)	Determined by the configured policy.
Take a snapshot.	O(log(h))	O(1)	Every time a component needs needs to be backed-up.
Commit generation.	O(b)	O(b)[4]	When snapshot process is completed.
Begin recovery procedure.	O(b)	O(b)	When the system boots.
Recover a component.	O(log(h))	O(1)	For each component that needs to be recovered.
Finish recovery.	O(1)	O(1)	When all components have been recovered.
Purge a generation.	O(b)	O(b)	Determined by the configured policy.
Purge zombies.	O(z * s * p)	O(1)	At most once per purge.

[4] O(b) because the entire `Snapper::Archive`, which contains all the backlinks, needs to

be written to the archive file.

8 Remarks on the Implementation of Snapper

Snapper should be able to be implemented via the Genode's API and provided data structures and the lwext4 library[5]. I was unable to get Genode's libc to work with PhantomOS so unfortunately libc is not viable for Snapper. Having this constraint in mind, here are what Snapper was optimized for:

8.1 Fast Insertion of Data

Insertion of new data during the snapshot procedure is relatively fast. All that is needed is to compute the hash of the data and to write both the data and its hash into a file.

8.2 Flexible Data Redundancy

Snapper allows the set of the data redundancy by allowing a file to have redundant copies (i.e. backlinks) after its reference count meets or exceeds `Snapper::Config::redundancy`. The archive file then links the virtual component to a comma separated list of files from older generations that store identical data. This redundancy comes at the cost of the following:

- slower insertions (due to additional string manipulations)
- higher disk usage (due to archive entries having longer strings)

It is important to note, however, that these costs are minimal and furthermore there are no costs pertaining to recovering data, as the Recovery Step tries to use the first file path provided by the archive entry. It accesses subsequent backlinks only if the first file was corrupted.

8.3 Fast Recovery of Generations

Recovery of entire generations comprises reading all files needed by the generation and loading the data into the address space. By using an array to keep track of where a component's file is located, Snapper can efficiently retrieve the data.

[5] `https://codeberg.org/jws/genode-wundertuete/src/branch/sculpt-24.04-2024-04-19`

8.4 Manageable Directory Sizes

Since each component on the address space needs a snapshot file, the performance would be be hampered severely if all those files in the same directory. By using a radix trie with a dynamic height, the files are distributed in a manageable way along the different directory levels, thus reducing the strain on the filesystem.

8.5 File and Generation Integrity

By utilizing hash, Snapper can detect when a snapshot file or archive has been tampered with. The admin of the system can then decide what to do with that knowledge through configuration of the policy.

8.6 Transient Files

Snapshot files that are relevant for more than one generation are not duplicated. Instead, the archive file keeps track of which files are needed for the generation, even if some of those files could be from other generations. Each file's reference count makes sure that a file is not removed while it is still needed by a valid generation. Similarly, files and directories that are no longer needed can easily be identified and removed, ensuring that storage space remains uncluttered.

8.7 Partial Snapshot Recovery

A generation with a valid archive, can be indexed to load a particular version of a component without having to restore the entire system to that generation.

9 The Snapper Object

The following section explains the usage of the main interfaces of the `Snapper` object. For using Snapper in an actual project, see

> The code in this section requires the snapper.h header. Error handling has been omitted for brevity.

9.1 Initializing Snapper

This step is **required** to use any functionality of Snapper. You can initialize the global object with:

```
Snapper::Main snapper(env);
```

Here `env` is the `Genode::Env&` object created at the start of the Genode program.

9.2 Creating A Snapshot

Make sure Snapper is initialized.

1. Prepare the Snapper object for the snapshot procedure.

   ```
   snapper.init_snapshot(); // OPTIONAL pass in specific generation
   ```

2. For each component's data that should be saved in the snapshot.

   ```
   int payload = 5;
   Genode::size_t size = sizeof(payload);

   Genode::uint64_t identifier = 4;

   snapper.take_snapshot(&payload, size, identifier);
   ```

3. Finally save mark the snapshot as complete and cleanup.

   ```
   snapper.commit_snapshot();
   ```

9.3 Restoring A Snapshot

Make sure Snapper is initialized.

1. Begin restoration procedure.

   > If no generation is provided to this method, the latest generation will be used for the restoration. If Snapper is to restore a specific generation, the caller should provide the RTC timestamp of the generation as a string.

   ```
   snapper.open_generation(); // OPTIONAL pass in specific generation
   ```

2. Restore each desired component (idenitified by its identifier).

   > The caller is responsible for providing a buffer sufficient for the data to be restored to.

```
Genode::size_t size = 5;
char data[size];

Genode::uint64_t identifier = 5;

snapper.restore(&data, size, identifier);
```

3. Cleanup the restoration process.

```
snapper.close_generation();
```

9.4 Purging A Snapshot

Make sure Snapper is initialized.

1. Purge a desired generation.

   > Provide a RTC timestamp as a string to delete a specific generation. By default the oldest generation is removed.

```
snapper.purge(); // OPTIONAL pass in specific generation
```

2. Purge expired generations

   The expiration for generations can be set in Configuration.

```
snapper.purge_expired();
```

3. Purge zombie files

```
snapper.purge_zombies();
```

10 Snapper User Stories (i.e Using Snapper in Projects)

Snapper is designed with Genode's RTC server-client paradigm in mind. You can find a demo configuration of the server in /run/snapper-common.inc. For configuring the server, see Configuration.

For the using the client (you can find an example in /src/test/snapper/main.cc), you need:

```
#include "snapper_session/connection.h"

void
Component::construct (Genode::Env &env)
{
  Snapper::Connection snapper (env); // establishes connection to the server
}
```

You can then use `snapper` as explained in The Snapper Object.

11 Handling of Fail Points

Here's how Snapper will handle the following failure points:

11.1 Improper Unmount of the File System

If the system were to crash then the filesystem would not be properly unmounted. This is already handled by the lwext4 library. On mount, it first tries to fix the filesystem. If that is unsuccessful it prints out a message that the fsck Linux utility should be used.

11.2 Incomplete Snapshot

In the case when the system crashes midway through a snapshot, the latest generation directory will still not contain an archive file. Thus when the system reboots the incomplete generation will be deleted and a prior valid generation will be used, if such exists.

11.3 Low Disk Space

If the system detects that disk space is running low, it will run the Purge algorithm on the oldest generation until either disk usage is back to a acceptable level.

12 Backups

Although Snapper can detect when snapshot files or archive files have been corrupted, it only supports redundancy when it comes to snapshot files (see the Data Redundancy section for more details). However, archive files can be corrupted as well. Snapper **does not concern itself with providing**

redundancy for the archive files. The reason for this decision is that there are many variables that a system administrator might want to tweak when backing up files that govern how a generation is to be recovered.

For instance, should the backup archive files be saved on a different disk? Should archive backup files from different systems be stored together? How should the backup files be named to differentiate them from one another?

With so many options and use cases, it is easier to leave the system administrator in charge of ensuring that the archive files are backed-up. If an archive file was deemed to have failed its hash, Snapper will notify the system administrator that the generation could not be recovered. All the system administrator has to do then is to simply replace the corrupted archive file with a backed-up copy.

> It is **highly recommended** to backup archive files! If an archive file were to be corrupted, disk storage would leak as some files will still have non-zero reference counts, even though there are no references to them from any valid archive files.

13 Configuration of Snapper

Snapper should be configurable through Genode's XML. The configuration options are stored in `Snapper::Config`:

OPTION	TYPE	DEFAULT	DESCRIPTION
verbose	`bool`	false	Whether to print verbose output.
threshold	`unsigned int`	100	The maximum number of files in a snapshot sub-directory.
integrity	`bool`	true	If true, crash the system on failed integrity checks, otherwise log a warning.
redundancy	`unsigned int`	3	After reaching this reference count, a redundant file copy will be created for subsequent snapshot.
max_snapshots	`unsigned int`	0	The maximum number of complete snapshots inside <snapper-root>.
min_snapshots	`unsigned int`	0	The minimum number of generations that need to be present for a purge to be possible.
expiration	`unsigned int` (seconds)	0	How many seconds a generation should be kept.

Continued on next page

Continued from previous page

OPTION	TYPE	DEFAULT	DESCRIPTION
bufsize	`size_t` (bytes)	1024 * 1024	The size of the dataspace which will transfer the payload to the snapper component.

13.1 Verbose Output

`Snapper::Config::verbose` (default = false) toggles verbose output.

13.2 Directory Load

`Snapper::Config::threshold` (default = 100) can be set to determine the maximum number of snapshot files within a snapshot sub-directory. After the number of files exceeds this threshold, an extender directory will be created and all subsequent files will be placed within said directory.

13.3 Integrity Checks

`Snapper::Config::integrity` (default = true), when true, will crash the system during the Recovery Step if a mapping in the archive file does not provide a single valid snapshot file. On false, Snapper will just log an error and ignore the restoration of that page.

Likewise when an archive file fails its hash check, the system will crash if `Snapper::Config::integrity` is set to true. Otherwise, an error will be logged and the system will boot without recovering that generation.

13.4 Redundancy Level

`Snapper::Config::redundancy` (default = 3) determines the maximum number of generations that a snapshot file appears in before another backlink is created.

13.5 Retention Policy

Snapper's retention policy will determine which *completed* generations are kept and which are purged.

13.5.1 Limit by Number

The number of completed generations kept will be limited to `Snapper::Config::max_snapshots`. This retention policy is disabled if that number is 0 (default). Conversely, the Purge Step will fail if the number of generations is lower than `Snapper::Config::min_snapshots` (default = 0).

13.5.2 Limit by Expiration

If a generation is older than `Snapper::Config::expiration` seconds it will be purged. This can be disabled by setting `Snapper::Config::expiration` to 0 (the default).

13.6 Payload Size

`Snapper::Config::bufsize` can be set according to the payload size expected for calls to `Snapper::take_snapshot()`. **Be aware** that calling `Snapper::take_snapshot()` with a payload larger than `Snapper::Config::bufsize` will result in a crash!

14 Conclusion

Snapper 2.0 efficiently manages PhantomOS snapshots by storing only changed data, reducing disk usage, and ensuring data integrity with ext4's logging features. It overcomes previous limitations with a robust mapping strategy and reference counting for file management. The dynamic directory structure enhances performance, while configurable retention policies and fail-safe mechanisms improve system resilience. Despite some challenges, Snapper provides a strong foundation for effective snapshot management in PhantomOS.