Technical Documentation

Problems in the Original Design:

Problems with the savevm architecture

Savevm essentially works the following way
a) Stop the currently running virtual machine
b) Save the current ram and device states to disk
c) Take a disk snapshot

While steps a and c take quite less time, step b is what slows everything down. This is because the time taken in step b essentially depends on how many pages of guest physical memory are written out to disk. So If more data is written out, the time for savevm is longer. Thus deciding on what to write out to disk is an important factor here.
The original savevm is so slow since in the worst case it writes out every page of the guest physical memory to disk every time a snapshot is taken. More details on why this is so are given below.

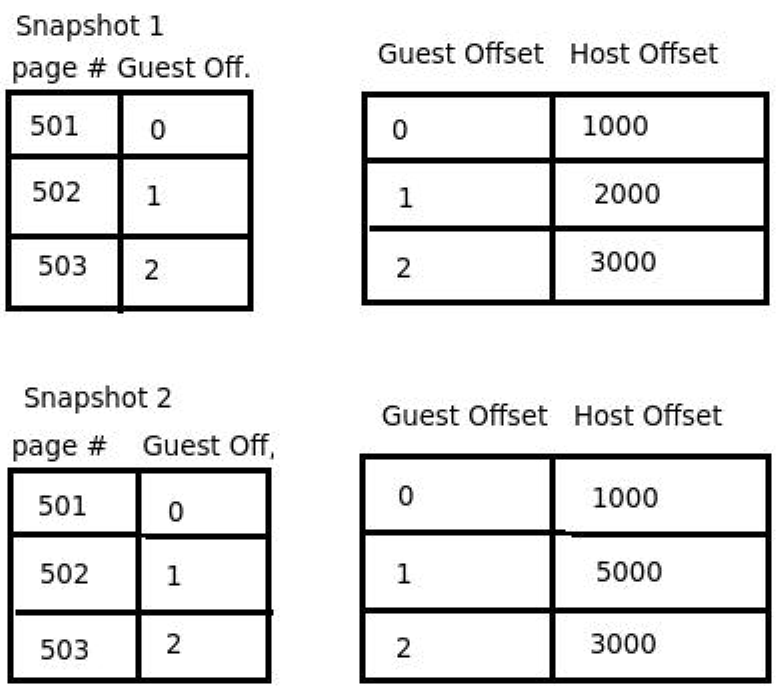
Qemu keeps track of pages which have gotten dirty since the last time the dirty flag was cleared. Once a page is written out during savevm, its dirty flag is reset. Lets say we took a snapshot at time t1 and are about to take another at time t2. Also say in the time interval t2 - t1, only a set 'delta' of pages have gotten dirty. In principle, if every other page is already on disk after we had taken snapshot at time t1, it is sufficient to save only the pages that have changed ('delta') while taking a snapshot at time t2. However qemu disregards this information when deciding on what to save. Infact its ram save code marks every page as dirty before deciding what to write. Since this information is lost, it just writes out everything

The simple solution here would be to just change this piece of code and not set all pages to dirty before writing them out. But a simple change like that results in incorrect behavior during loadvm. The reason for this is because there is a much tighter integration between the qemu code which sends out pages to be written and the qcow2 driver which manages where this data is written to.

When qemu decides to write out a page, it has a certain guest offset associated with it where the page contents should be written at. The guest offset initially starts at zero and is automatically incremented as more and more data is written. When the underlying qcow2 block driver finally writes this data out, it maps these guest offsets to corresponding host offsets in actual files. Therefore in a design where we save only dirty pages, we should have a way of correctly determine the mapping between our data and the host offsets they were written to previously. Since the guest offsets always start from zero every time we take a snapshot, the simple solution as mentioned above fails to preserve this mapping and hence is not effective.

For Example, the expected behavior is what is shown in figure1. Pages 501, 502 and 503 are incrementally assigned offsets 0,1 and 2 which are mapped to host offsets 1000,2000 and 3000 in snapshot 1. When snapshot 2 is taken, page number 502 has changed. It should still be mapped to offset number 2 which should be mapped to a different host offset in the actual file where the changed contents are written to. The host offsets corresponding to pages 501 and 503 should remain unchanged.

The actual behavior is however as shown in figure2 if we follow the simple approach of just writing out dirty pages without taking care of the page to offset mapping. Page 502 is changed and is written out. However, since qemu assigns guest offsets incrementally as pages are written out, 502 is assigned an offset 0. This causes the host offset corresponding to offset 0 to be remapped which is incorrect.



Snapshot 1

page #	Guest Off.
501	0
502	1
503	2

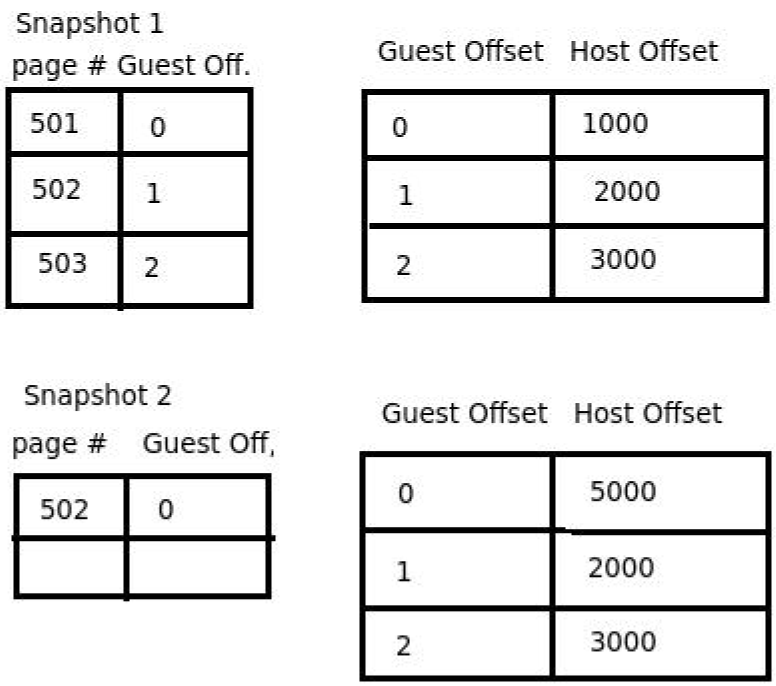
Guest Offset	Host Offset
0	1000
1	2000
2	3000

Snapshot 2

page #	Guest Off,
501	0
502	1
503	2

Guest Offset	Host Offset
0	1000
1	5000
2	3000

Figure 1: Expected Behavior while saving RAM state



Snapshot 1

page #	Guest Off.
501	0
502	1
503	2

Guest Offset	Host Offset
0	1000
1	2000
2	3000

Snapshot 2

page #	Guest Off,
502	0

Guest Offset	Host Offset
0	5000
1	2000
2	3000

Figure 2: Actual Behavior if the simple approach is followed

Problems with qcow2

The qcow2 driver makes use of four different kinds of tables to manage host clusters. Two of these tables are the L1 and the L2 tables which maintain the mapping between guest and host offset while the other two are refcount tables which maintain reference count for each host cluster.

Savevm functions in qemu put data to be saved in a buffer and flush it once its full. Every time the driver writes a buffer of data out, some of these tables might get updated. Since in qcow2, the cache is writethrough by default, every such buffer flush causes the changed tables to be flushed out as well. The default buffer size is 32 K while a L2 table occupies 64K (one host cluster) by default. So for every buffer flush of 32 K, there is an additional 64 K of data written out. Hence the total data written out during savevm is much higher than what it should be normally.

Modified Design:

Changes in the savevm architecture

The part of savevm design which handled which pages to save is now changed to save only dirty pages. 'Clean' pages are skipped and the guest offset is incremented accordingly to ensure that the mapping of pages to guest and host offset remains consistent.

A regular file named 'snap' is maintained by the qemu zfs plugin where this state information is written to. Since this is a regular file, the guest offsets being written to are exactly the same as offset into the file where the data is actually written. Hence it is easier to maintain the desired mapping.

The Qemu ZFS Plugin

A new block driver for images having a zfs filesystem and structured as mentioned in the installation section on the webpage (http://compas.cs.stonybrook.edu/downloads/qemu-snapshots/), has been added. This driver functions exactly as the raw image driver present in an original qemu installation except that it has snapshotting capabilities. It implements all the interfaces used by savevm functions to write the state out during savevm or read pages back during loadvm. Internally it executes zfs commands on the shell using the 'system()' function to create and manage snapshots.