Qemu - Incremental Checkpointing

By Nora Abi Akar and Marina Shimchenko

Qemu is a full-system functional simulator that PARSA is using to integrate with Flexus, a detailed microarchitectural simulator to create the simulation stack: QFlex. Full system simulation of multiprocessor systems is up to a million times slower than real hardware. This simulation time can be drastically reduced by exploiting the homogeneity of application performance and applying statistical sampling. The big challenge facing the realization of sampling lies in rapidly constructing the correct initial state for the large number of performance measurements; this is known as the warming problem. To solve this problem, warm architectural and microarchitectural states are stored in checkpoints. The checkpoints can be loaded in parallel, thus further reducing simulation time.

In developing QFlex, many modifications have to be applied to Qemu to make integration with Flexus feasible and to make the process of simulation easy and inexpensive. One of the changes introduced is making Qemu checkpoints (also referred to as snapshots) incremental so as to save storage space. This document describes the changes made to Qemu to make snapshots incremental and explains what the user needs to know to be able to use this feature.

Snapshots in the original Qemu

In the original Qemu v2.6.0. a “VM snapshot is a snapshot of the complete virtual machine including CPU state, RAM, device state and the content of all the writable disks”[1]. A snapshot can be taken using the monitor command “savevm” followed by the snapshot name. The disk image snapshots are stored inside the disk image. This is what we need for our application, however, Qemu saves the full content of the RAM in every snapshot. This quickly amounts to a huge storage space when we use it to solve the warming problem in our statistical simulation environment. The size of a snapshot can be greatly reduced if instead of saving every page in the RAM every time we create a snapshot, we only save the pages that have been modified since the last checkpoint, thus making the snapshots incremental.

We can have a broad understanding of snapshots in Qemu by looking at the files savevm.c and ram.c in the migration folder. Saving a snapshot happens in 3 stages:

setup, iterate and complete. The functions are called from savevm.c and defined in ram.c.

During the setup stage a migration bitmap of size equal to the number of pages in our RAM is created and filled with 1s. This bitmap is then synchronized with the list of dirty pages in the RAM (ram_list.dirty_memroy). The synchronization process performs a bitwise or of the migration bitmap and ram_list.dirty_memory, saves the resulting bitmap in the migration bitmap, and resets ram_list.dirty_memory. During the iterate stage the migration bitmap is traversed and when a 1 is encountered it is set to 0 and its corresponding RAM page is saved alongside its RAM address. During the complete stage, the migration bitmap is resynchronized with tram_list.dirty_memroy and the same process of the iterate stage is repeated. Finally the migration bitmap is deleted.

This process may seem odd at first sight. If the migration bitmap is initialized to all 1s, the synchronization stage appears pointless. And since the process of saving the snapshot starts with pausing the machine, there doesn't appear to be a need to resynchronize the bitmap. However, the process of saving a snapshot seems to be combined with the live migration function in Qemu and this can explain the peculiarities[1] .

The live migration feature in Qemu is described as "A postcopy implementation that allows migration of guests that have large page change rates (relative to the available bandwidth) to be migrated in a finite time. VMs of any size running any workload can be migrated. Postcopy means the VM starts running on the destination host as soon as possible, and the RAM from the source host is page faulted into the destination over time. This ensures there is a minimal downtime for the VM as compared to precopy, where the migration can take a lot of time depending on the workload and page dirtying rate of the VM"[2]. I believe that as part of live migration, a first copy of the RAM is sent from the source machine to the destination machine, while the source machine is still running. When the RAM contents have been migrated, the source machine is paused, and whatever pages have been modified since the RAM was migrated are copied to the destination machine. This explains the need for a migration bitmap and the two instances of synchronization.

Regardless of the reasons why the migration bitmap is used in the original version of Qemu, it provided a basis for our changes to make the snapshots incremental.

[1] Please note that this is just my interpretation of the code. I have not tested the live migration feature and cannot be sure that my thought process is correct.

Incremental snapshots in Qemu

To implement incremental snapshots in Qemu we had to establish the new logic behind saving and loading a snapshot.

1. Save a snapshot:
    - If this is the first snapshot we take after booting, with no previously loaded snapshot, save the entire contents of the RAM. This snapshot will provide the base copy of the RAM.
    - If a snapshot has already been saved/loaded before the current save operation, save only the pages that have been modified since the last load/save.
    - Always save information that allows us to load RAM pages in their appropriate location
    - Save the list of dependencies for each snapshot.

2. Load a snapshot:
    - Retrieve the dependency list of the called snapshot.
    - Load all snapshots in the list starting from the base RAM snapshot to the last snapshot in the list (the called snapshot).
    - When loading a snapshot, the RAM pages have to be loaded in their correct location in the RAM.

To accommodate the new functionalities, some of our changes are described. The full code should be referred to for further details. Our first goal was to take an “incremental” snapshot, meaning a snapshot that does not save the entire contents of the RAM. To do this, we simply filled the migration bitmap with 0s instead of 1s upon creation. This works because upon booting a machine, the RAM is set as all dirty ensuring that ram_list.dirty_memroy is all 1s. Therefore, the first snapshot after booting will save a full copy of the RAM after synchronizing the migration bitmap. After saving that initial snapshot, the ram_list.dirty_memroy is reset to all 0s and then individual pages are set to dirty (1 in their position in ram_list.dirty_memroy) when they are written. This implies that consecutive snapshots will only save dirty pages.

After we were able to save a small incremental snapshot, we had to load it and ensure that we do not break operation. Because in the original Qemu snapshots save the full content of the RAM, we were not sure that loading an incremental snapshot will guarantee that saved pages are loaded in their correct place in the RAM. However, we found that Qemu indeed loads the RAM pages in their correct location regardless of whether we have a full or incremental snapshot.

This suggested that loading a base snapshot followed by an incremental snapshot should preserve the state of the machine and allow normal operation to continue. But this was not the case. We had an error related to loading the attached devices. The cause of the error was a function call in the beginning of a snapshot load that resets all devices. This function accesses some read-only pages in the RAM and clears their content. If these pages are not modified between snapshots, they are not saved, but the load function still expects them to be loaded with every snapshot and reports an error when they are not. To fix the issue we decided to save these read-only pages of the RAM in every snapshot. We were able to load an incremental snapshot after applying the change. However, this meant that the minimum size of our incremental snapshot is 12Mb for an aarch64 machine. To further reduce the size of the snapshot, we instead disabled the resetting of this area of the RAM in the load function and we were able to obtain incremental snapshots where the size is in the order of Kb.

Once the basic operation of saving and loading was performed and tested we had to create and manage the dependency lists of snapshots to allow for automatic saving and loading of snapshots. Every time we save a snapshot the full list of snapshots it depends on should be saved alongside it, in the correct order, and referred to when we load said snapshot. The following process was followed in building the list of dependencies.

- A linked list is created (in savevm.c) to keep track of our snapshot dependency list.
- When a snapshot is saved, its name is pushed into the list. A text file “dependency_list” is opened and the name of the snapshot is printed inside to indicate that the list of snapshots that follow belong to that snapshot name and should be loaded before it. Then the linked list is traversed and all the snapshots names inside are printed in the file.
- When a snapshot is loaded, the file “dependency_list” is opened and searched for a line where the first word read is the name of our snapshot. Once this line is found; the linked list is cleared; the names of the snapshots in the line are read; and one by one the snapshots are loaded and their names pushed in the clean list.

This process guarantees that we always have a correct view of our snapshot and its dependencies internally in Qemu and externally as a textual representation of the snapshot dependencies. The below diagram illustrates this process more clearly.

Changes to user interface

We have added some options to make using the new snapshots easy for the user.

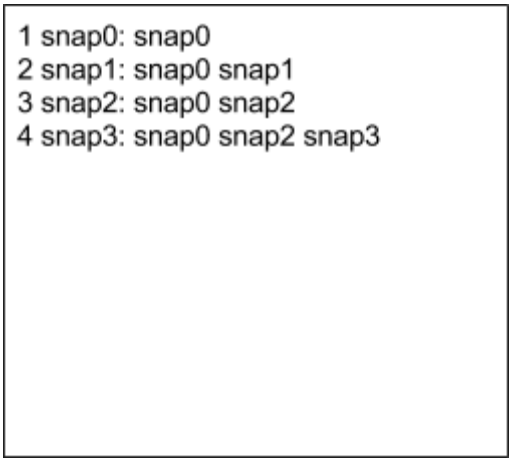
1. The dependency_list file:
   By default a text file called dependency_list is created/opened in the current directory and used for the snapshot saves and loads. If we want to save several

dependency_lists in the same directory, or we want to change the location/name of the file, we can specify the exact path to the file in the monitor commands: savevm loadvm and delvm.

Example: savevm snap1 - saves snap1's dependency list in ./dependency_list

savevm "snap1 ./home/qemu_checkpoints/arm_list.txt" - saves snap1's dependency list in ./home/qemu_checkpoints/arm_list.txt

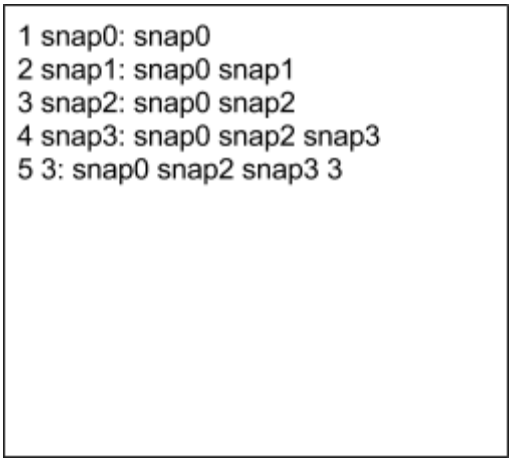
2. Deletions/Replacements:
   In the original Qemu, saving a snapshot with a name already in use for another snapshot replaces the old snapshot. We extend this functionality to our version. However, deleting/replacing a snapshot in our Qemu can cause a problem if not performed correctly. Deleting a snapshot that has other snapshots that depend on it will delete that snapshot and all its dependent snapshots. Therefore we do not allow this deletion/replacement unless explicitly requested.
   Example: delvm snap0 - aborts and prints a warning message
   delvm "snap0 FORCE" - deletes snap0 and all its dependents.
3. You may use snapshot id or snapshot name for delvm and loadvm. But be careful to save snapshot with numeric name, because in this case it can be loaded only using id.



So if you have such list of snapshots in your dependency file, then following command would be the same:

delvm "snap0 FORCE" → delvm "1 FORCE"
loadvm snap2 → loadvm 3

But if you try to save snapshot with numeric name, for instance "3", you get



And now every time you try to load snapshot with name “3”, snapshot with name “snap2” would be loaded, because it has id number “3”.

Testing

To verify that our changes do not break any functionalities in Qemu, we used an aarch64 machine and ran memcached for 4000 seconds while taking a snapshot every 10 seconds. The workload ran smoothly. When loading any of the 400 snapshots, Qemu resumed work correctly. We also tested deletion, replacement and booting directly from a snapshot.

Also it was made some test for measuring time and size of snapshots. It was used the workload as described before, but with different parameters of size of memory and percentage of reads/writes requests. During communication between client and server, chain of snapshots was saved and save time for each snapshot was printed in the output file. Then average value was calculated. After process was finished, last snapshot was loaded many times, also followed by printing of time required for loading snapshot. Then average load time was calculated.

Tis - average save time for incremental snapshots (new version of snapshots)
Til - average load time for incremental snapshots (new version of snapshots)
Tos - average save time for original snapshots
Tol - average load time for original snapshots
Zi - average size of incremental snapshots (new version of snapshots)
Zo - average size of original snapshots

Qemu image 1G, server memory 4G:

./loader -a ../twitter_dataset/twitter_dataset_unscaled -o ../twitter_dataset/twitter_dataset_5x -s docker_servers.txt -w 4 -S 5 -D 512 -j

./loader -a ../twitter_dataset/twitter_dataset_5x -s docker_servers.txt -g 0.8 -c 200 -w 4 -e -r 1000 -t 4000 -T 1

Results:

Tis = 0,12591s Zi = 9,36M
Til = 13,23s Zo = 997M
Tos = 6.32s
Tol = 10,2s

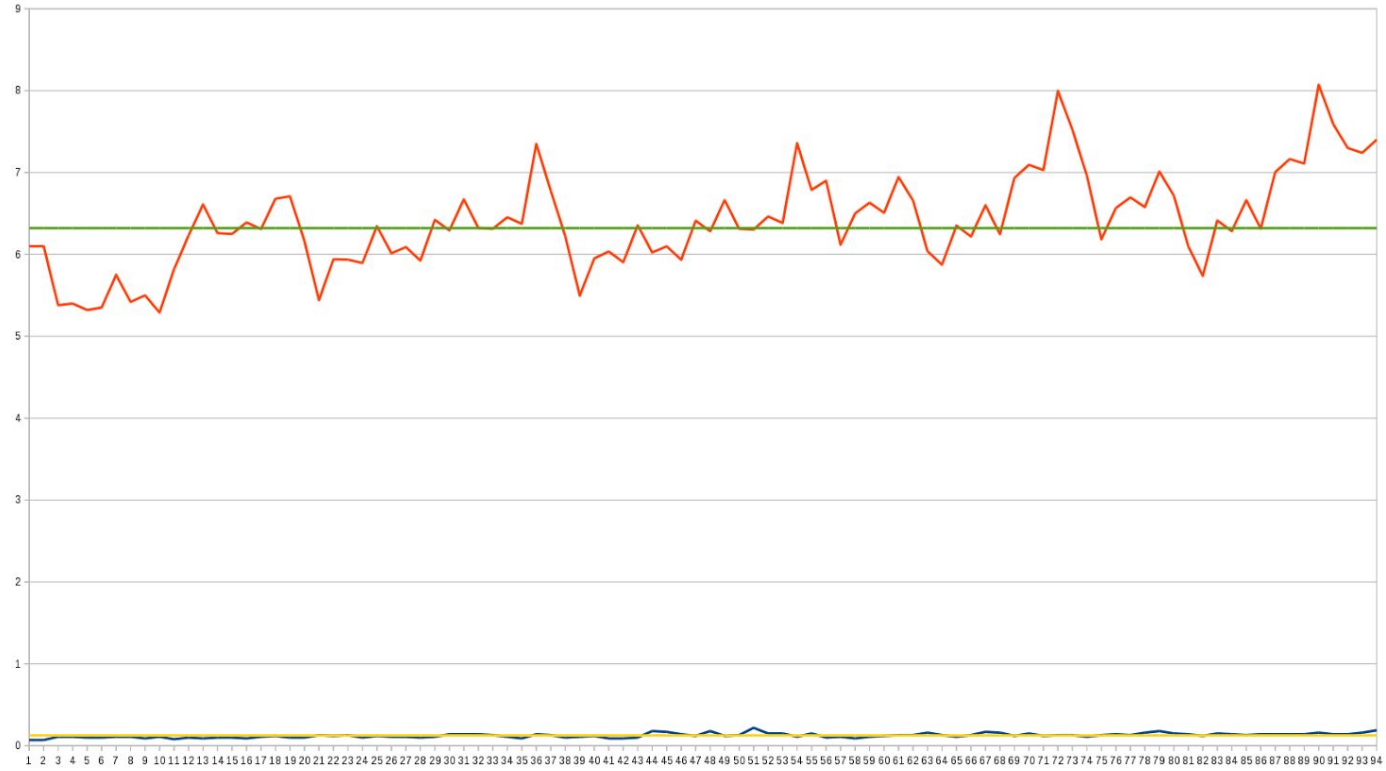


Chart 1. Comparing save time for original snapshots and incremental. By horizontals: snapshots number. By vertical: seconds.

- save time for original snapshots
- average save time for original snapshots
- save time for incremental snapshots
- average save time for original snapshots

Qemu image 10G, server memory 4G:

./loader -a ../twitter_dataset/twitter_dataset_unscaled -o ../twitter_dataset/twitter_dataset_5x -s docker_servers.txt -w 4 -S 30 -D 512 -j

./loader -a ../twitter_dataset/twitter_dataset_5x -s docker_servers.txt -g 0.8 -c 200 -w 4 -e -r 1000 -t 4000 -T 1

Results:

Tis	= 0,186s	Zi	= 14M
Til	= 62,16s	Zo	= 4G
Tos	= 35,54s		
Tol	= 45,98s		

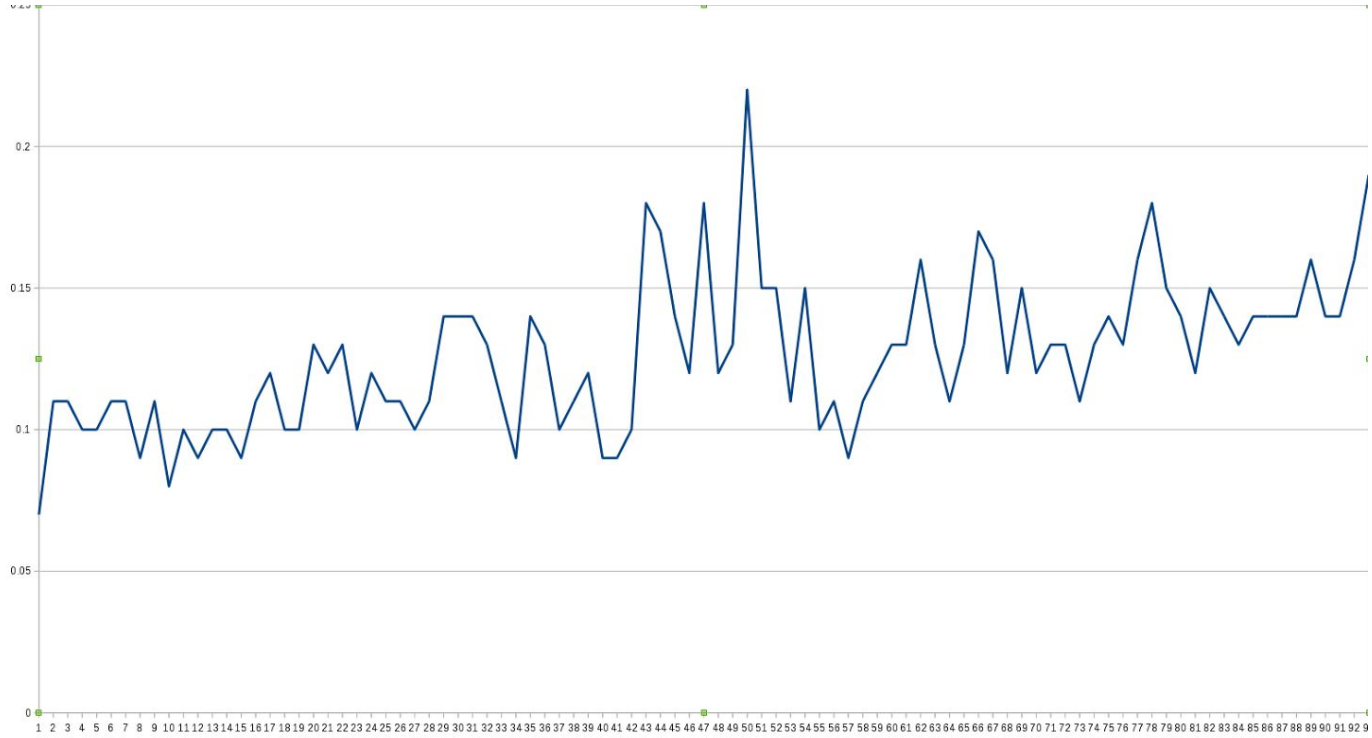


Chart 2. Save time for incremental snapshots for 1G image. By horizontals: snapshots number. By vertical: seconds.

Qemu image 10G, server memory 4G:

./loader -a ../twitter_dataset/twitter_dataset_unscaled -o ../twitter_dataset/twitter_dataset_5x -s docker_servers.txt -w 4 -S 30 -D 512 -j

./loader -a ../twitter_dataset/twitter_dataset_5x -s docker_servers.txt -g 0.5 -c 200 -w 4 -e -r 1000 -t 4000 -T 1

Results:

Tis = 0,296s Zi = 18M
Til = 67,09s Zo = 4G
Tos = 35,54s
Tol = 45,98s

Qemu image 10G, server memory 4G:

./loader -a ../twitter_dataset/twitter_dataset_unscaled -o ../twitter_dataset/twitter_dataset_5x -s docker_servers.txt -w 4 -S 30 -D 512 -j

./loader -a ../twitter_dataset/twitter_dataset_5x -s docker_servers.txt -g 0.2 -c 200 -w 4 -e -r 1000 -t 4000 -T 1

Results:

Tis = 0,312s Zi = 22M
Til = 70,01s Zo = 4G
Tos = 35,54s
Tol = 45,98s

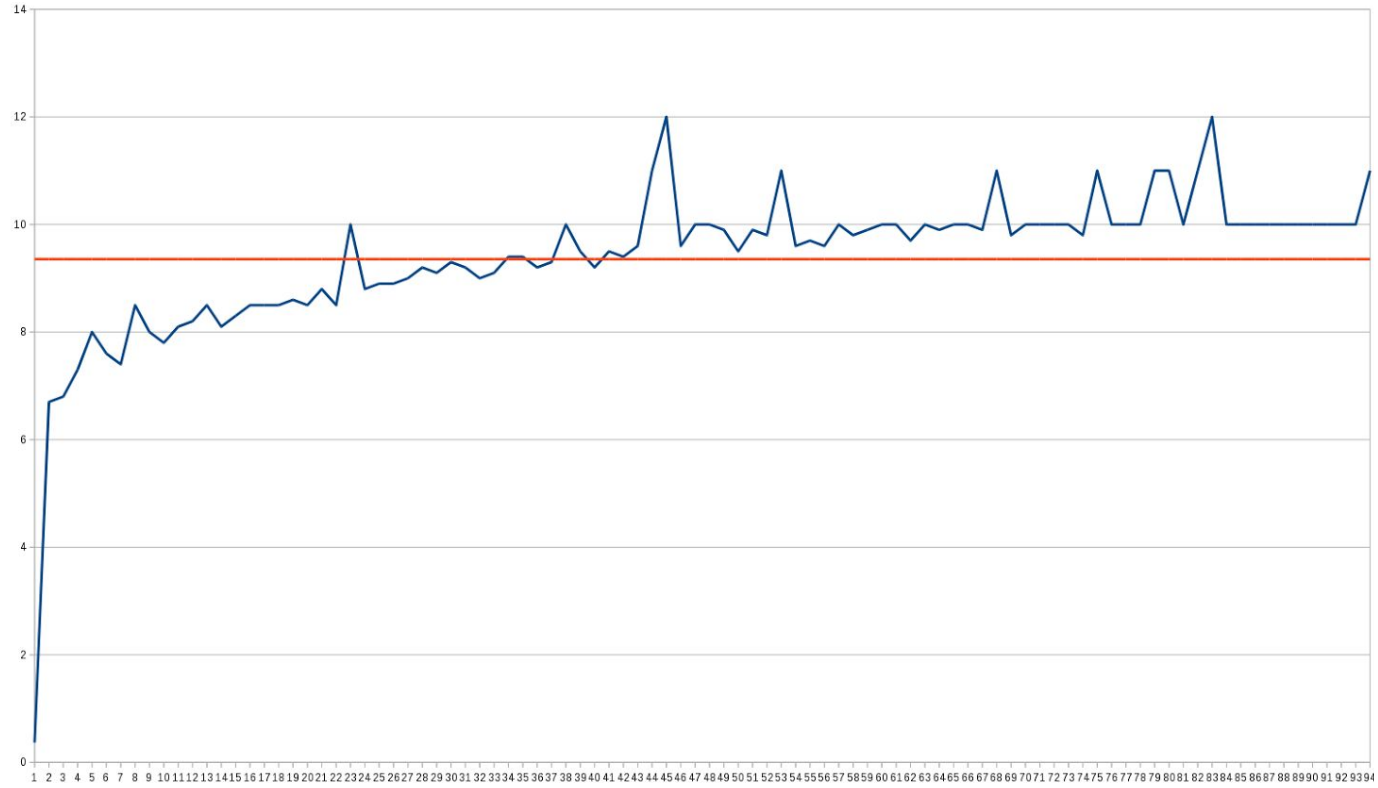


Chart 3. Size for incremental snapshots. By horizontals: snapshots number. By vertical: Mbytes.

average size of incremental snapshots

 size of incremental snapshots

Bugs:

This problem was found at 20:03 on Friday, which was the last work day for intern, who was working on it. Now only images with the size multiplied size of qemu page (1024) works. The problem is on the load side. In the function ram_list_clean, which is called from ram_load(), located in migration/ram.c.

References:

[1] http://wiki.qemu.org/download/qemu-doc.html
[2] http://wiki.qemu.org/Features/PostCopyLiveMigration