Programming Support for Autonomizing Software: Technical Report

Anonymous Author(s)

1 Checkpointing/Restore

In reinforcement learning, when entering the ending state (e.g., Mario dies), current training procedure will stop until current state is restored back to the initial state (e.g., Mario begin). Then the training continues with the new initial state. In order to support such mechanism in any program, primitives *au_checkpoint*() and *au_restore*() help target program checkpoint/restore arbitrary program states.

While the developer can put the primitive *au_checkpoint*() in a program point to indicate his/her intention of creating a checkpoint. The primitive may be present inside a loop (e.g., the game loop) so that many redundant checkpoints are created. To avoid such redundancy, we require the user to press a hot key during software execution. A checkpoint is created only when both a hot key impression is present in the event queue and the checkpointing primitive is encountered during execution. This is analogous to pressing "*save-game*" during game play.

As mentioned earlier, we only checkpoint software states and database states but not model states. However, this is difficult to achieve as all these states are in the process space and indistinguishable for KVM. As such, before restoring to a checkpoint, Autonomizer saves the current model states to persistent storage. After restoring, it overwrites the model states with those saved in storage.

We designed a client-server protocol for supporting arbitrary program state checkpointing/restoring. The target program for autonomization runs on the guest OS as the client. On the other side, a checkpoint server runs on the host OS and listens to the request from the client process and the user. The checkpoint server is in charge of client states checkpoint/restore and model files backup.

Protocol Requests. The requests issued during the protocol execution is listed in Fig. 1 and the following describes the runtime execution flow about checkpoint/restore.

Protocol Requests :
@CKP | @RES | @YES | @NO | @GET | @SET

Figure 1. Requests

Checkpoint. In Fig. 2, the checkpoint server repeatedly listens to the checkpoint request *CKP* from the client and the user. If the user decides it is the right time to checkpoint current program state, he/she then sends the *CKP* request (① in Fig. 2) to the checkpoint server through keyboard. The checkpoint would record the request from the user and listen to the request from the client side program. Whenever the client program execution reaches the primitive *au_checkpoint*, it issues *CKP* request (② in Fig. 2) to the checkpoint server to check any checkpoint request from the user. Because the checkpoint server received the *CKP* request from both the user and the client, it begins to make a snapshot of the virtual machine to checkpoint client's current state and the client then continues its execution right after receiving *YES*, the confirmation message (③ in Fig. 2). On the other hand, if there is no checkpoint request from the user, the checkpoint server just sends the *NO* message back and the client process then continues its execution until its next reaching of *au_checkpoint*().

Restore. In Fig. 3, whenever the client program enters the ending state (e.g., Mario dies), it invokes the primitive *au_restore*. It first issues the *SET* request (① in Fig. 3) to send the up-to-date model and metadata to the checkpoint server for backup. After sending all local files, the client program then issues the *RES* request (② in Fig. 3) for program state restoring.

On receiving the *RES* request, the checkpoint server first checks whether there is any existing snapshot made previously. If yes, the whole virtual machine including the client program will be restored to the previous checkpointed state. The client program then continues its execution from the checkpointed location (③ in Fig. 2). The client program then issues the *GET* request (④ in Fig. 2)to the checkpoint server in order to get the backed up model and metadata. The checkpoint server then issues the *SET* request (⑤ in Fig. 2 to send all files back to the client such that the client program can continue the training with up-to-date model before restoring.

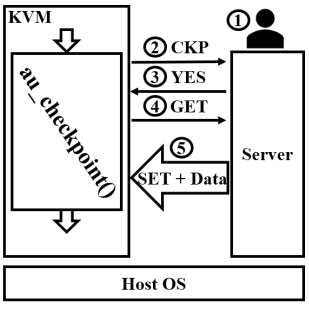

Figure 2. Checkpoint

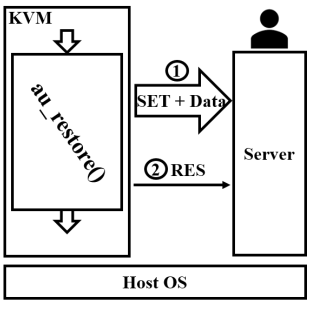

Figure 3. Restore

2 OpenTuner For Canny

Autonomizer enables online prediction (Section 1.1), which is much faster (~1.5s/image) than autotuning when deployed. As far as the OpenTuner tuning results for Canny is concerned, we show it in the following table.

Table 1 shows the tuning results of 10 images for Canny using OpenTuner. Columns 2 and 3 represent the SSIM score of each image after running OpenTuner for 5 and 10 seconds respectively. Column 4 represents the Autonomizer SSIM score. We ran each experiment 10 times and took the average.

Table 1. OpenTuner experimental results

	OpenTuner(5s)	OpenTuner(10s)	Autonomizer(~1.5s)
Image 1	0.75	0.91	0.83
Image 2	0.81	0.85	0.87
Image 3	0.78	0.89	0.90
Image 4	0.74	0.88	0.91
Image 5	0.61	0.67	0.63
Image 6	0.47	0.68	0.57
Image 7	0.58	0.79	0.82
Image 8	0.52	0.73	0.57
Image 9	0.59	0.67	0.63
Image 10	0.70	0.86	0.88
Average	0.655	0.793	0.761