VM Migration and Replication

Yiying Zhang
Outline

• VM migration: “Live Migration of Virtual Machines”

• VM replication: “Remus: High Availability via Asynchronous Virtual Machine Replication”
The Need for VM Migration

- Load balancing
  - Move VMs from heavily loaded to less loaded host machines
  - Move VMs to new machines added to the cluster
- Recovery from host failure
  - Move (restart) VMs on a new host when its original host fails
- Maintenance of host
  - Temporarily move VMs off a host when this host is under maintenance
Types of Migration

- Whole VM migration
- Process migration
- Cold migration: shutdown a VM and restart it on a new host
- Warm migration: suspend a VM, move its states, resumes on a new host
- Live migration: move a VM’s states while it continues to run (with a brief stop)
Live VM Migration

• Migrate a running VM (without or with minimal disturbance to its applications)
  • All its in-memory, network, storage states

• Benefits
  • After migrating the whole VM, nothing left at the old server
  • In-memory states can be transferred consistently (and efficiently)
  • Allows a separation of concerns b/w the users and operator
Goals of Live VM Migration

- Minimize *downtime*
  - The time when services provided by VM are unavailable
- Small total *migration time*
  - The total time taken for all VM states to be migrated
- Migration has minimal effect on foreground services
  - Minimal resource contention with foreground VM services
Naive Migration: Checkpoint, Move, and Resume

• Pause the VM and take a snapshot of it
  • Record CPU registers, memory content, I/O states, file system
• Move the snapshot to the new machine
• Restore the snapshot
  • Restore in-memory states
• Resume the VM execution
Naive Migration Performance

- Time to move memory to memory + time to move disk/SSD to disk/SSD

- e.g., 10Gbps network, 120 MB/s disk sequential read/write bandwidth (7200 RPM disk), 17 GB/s (DDR4)
  
  - Migrating a VM with 16 GB memory and 32 GB disk
  
  - Time to move disk data: \( \frac{32 \text{ GB}}{120 \text{ MB/s}} = 4.55 \text{ min} \)
  
  - Time to move memory data: \( \frac{16 \text{ GB}}{10 \text{ Gbps}} = 12.8 \text{ sec} \)

  - Plus the time to stop and resume the VM execution

  ➡ Downtime at least 4.6 min
Improving Storage Migration

• Use disaggregated storage (e.g., EC2, EBS)

• No need to move storage when migrating VMs

• What about local storage?
  • Only copy deltas of a base image?
  • Don’t copy everything all together?
  • Replicate local storage on multiple nodes (for reliability but could also benefit migration)
Improving Memory Migration

- Even if we don’t need to move storage, moving memory with the naive approach would still cause too much downtime (e.g., 12.8s in our example)

- The focus of this paper

- This paper’s approach: *pre-copy* migration
Memory Migration Phases

• Push phase
  • Source VM continues running while pushing memory to new VM

• Stop-and-copy phase
  • Stop the source VM, copy all (remaining) memory to new VM, start new VM

• Pull phase
  • After new VM starts, pull memory from source VM as needed (accessed)

• Tradeoffs between these options?
Pre-Copy Migration

- Iteratively pre-copy (modified memory)
  - All memory copied in the first round
  - $n$ round copies the modified memory content of round $n-1$
- A final step of stop-and-copy follows the pre-copy phase
  - This step determines downtime
- When shall we stop the pre-copy phase and move to stop-and-copy?
Writable Working Set

• For memory that is never/rarely modified (or at least not/rarely modified during the migration time) => pre-copy

• For memory that is frequently modified, i.e., *Writable Working Set (WWS)* => stop-and-copy

• Goal: estimate WWS to minimize the memory need to go through stop-and-copy (without making the pre-copy phase too long (affects total migration time))

• Could use heuristics or more smart ways to determine WWS
Figure 2: WWS curve for a complete run of SPEC CINT2000 (512MB VM)
Migrating Network State

• Goal: maintain all open network connections after migration without redirection

• Environment: migration within the same LAN

• Solution:
  
  • VM carries the original IP address and all protocol state
  
  • Generate unsolicited ARP (Address Resolution Protocol) reply from the migrated host to other nodes, advertising the new physical address of the IP
  
  • Other nodes reconfigure to send to the new physical address
Figure 1: Migration timeline

Stage 0: Pre-Migration
Active VM on Host A
Alternate physical host may be preselected for migration
Block devices mirrored and free resources maintained

Stage 1: Reservation
Initialize a container on the target host

Stage 2: Iterative Pre-copy
Enable shadow paging
Copy dirty pages in successive rounds.

Stage 3: Stop and copy
Suspend VM on host A
Generate ARP to redirect traffic to Host B
Synchronize all remaining VM state to Host B

Stage 4: Commitment
VM state on Host A is released

Stage 5: Activation
VM starts on Host B
Connects to local devices
Resumes normal operation

VM running normally on Host A

Overhead due to copying

Downtime (VM Out of Service)

VM running normally on Host B
Migration Mechanisms

- Managed migration
  - Migration performed by management VM (e.g., Dom0), guest OS agnostic
  - Needs to track dirty pages during rounds of pre-copy
  - Using an initially read-only shadow page tables in Xen to capture writes
- Self migration
  - Guest OS handles most of the migration
  - Guest OS tracks dirty memory and pre-copies them
  - Uses a two-stage stop-and-copy phase
Other Implementation Details

• Dynamic rate-limiting
  • Dynamically adapt the network bandwidth to balance between good migration performance and low impact on foreground activities

• Rapid page dirtying
  • Avoid transferring pages that are repeatedly dirtied

• Paravirtualized optimizations
  • Run a thread in VM to monitor the WWS of guest processes
  • Let guest OS free less useful memory (e.g., cold buffer cache pages)
Results

Effect of Migration on Web Server Transmission Rate

Throughput (Mbit/sec)

Elapsed time (secs)

512Kb files
100 concurrent clients

870 Mbit/sec
1st precopy, 62 secs

765 Mbit/sec

694 Mbit/sec

9.8 secs

further iterations

165ms total downtime

Sample over 100ms
Sample over 500ms

Figure 8: Results of migrating a running web server VM.
Results

Iterative Progress of Live Migration: SPECweb99
350 Clients (90% of max load), 800MB VM
Total Data Transmitted: 960MB (x1.20)

Area of Bars:
- VM memory transferred
- Memory dirtied during this iteration

The first iteration involves a long, relatively low-rate transfer of the VM’s memory. In this example, 676.8 MB are transferred in 54.1 seconds. These early phases allow non-writable working set data to be transferred with a low impact on active services.

In the final iteration, the domain is suspended. The remaining 18.2 MB of dirty pages are sent and the VM resumes execution on the remote machine. In addition to the 201ms required to copy the last round of data, an additional 9ms elapse while the VM starts up. The total downtime for this experiment is 210ms.

Figure 9: Results of migrating a running SPECweb VM.
Other VM Migration Systems

- VMware vMotion
- XenMotion
- Migration across heterogeneous architecture
- Migration across datacenters
- Migration from one cloud provider to another provider
Outline

• VM migration: “Live Migration of Virtual Machines”

• VM replication: “Remus: High Availability via Asynchronous Virtual Machine Replication”
Benefits of VM Replication

• Failures are a norm in datacenters

• Datacenter and cloud applications all need high availability and reliability

• both achieved by adding redundancy

• Usually, software systems/services implement their own redundancy schemes (e.g., storage replication, redundant metadata server, etc.)

• But if we could replicate the whole machine, we can provide redundancy to all applications running on it
Remus: High Availability via Asynchronous Virtual Machine Replication

CS 736 - 4/13/17
Hypervisor-Based High Availability

Big Idea:

- Run unmodified software (not fault tolerant)
- Run commodity server hardware (not fault tolerant)
- Use the hypervisor to transparently replicate state
- Goal: all observable output should be identical to running a single machine that does not fail
Hypervisor High Availability Strawman #1

Run app in a hypervisor using reliable shared storage:

● Restart elsewhere on crash
  ○ Just like local boot

● Live state is lost during crash

● Not transparent
Hypervisor High Availability Strawman #2

Run two VMs and duplicate all inputs to both VMs (identical inputs to both VMs)

- VM states will be identical if computation is deterministic

Challenge: determinism is hard

- Interrupts
- Multiprocessors
- Cache consistency
- Timestamps
Hypervisor High Availability Strawman #3

- Run a primary VM and a backup VM
- Only run compute on the primary VM
- Periodically replicate complete system state to the backup VM
- Block all output for an epoch until replication completes

This is the Remus approach!
Remus Overview

1. Run a primary app inside a VM
2. Periodically pause VM and send a snapshot to a backup
3. All output from primary is buffered until the snapshot completes
   a. Primary does not need to pause after every output (keeps executing)
4. Disk writes are immediately propagated to backup
   a. Disk writes are buffered until the next memory snapshot
5. The backup VM does not execute
Remus Implementation: Leverage Live Migration

- Traditional live migration
  - Stop VM at primary
  - Migrate state to backup
  - Start execution at backup
    - Spurious ARP to reprogram the network

- Remus migration
  - Pause VM at primary
  - Migrate state to backup
  - Restart VM at primary
Hypervisor High Availability Challenges

Primary challenge: Performance!

- VM memory replication
- Output blocking (delay)
- Frequent pausing (delay)
  - E.g., Do op -> replicate -> ACK -> release
  - Network latency bounds performance if the primary cannot execute ahead (asynchronously)
- Large data volume to replicate
  - E.g., Rehashing is a small operation that can change lots of data
Remus Solution to Deliver Good Performance

- The primary executes *speculatively*, then checkpoints and replicates its states *asynchronously*

- System states are visible externally only after the checkpoint is committed
Remus Approach

• Speculative execution

  • The primary buffers outputs and releases them to the outside world later

  • It executes before replication finishes (which is the point where a failure can be sustained)

• Asynchronous replication

  • The primary VM resumes execution right after it makes a checkpoint (which captures all the changes in the past epoch)

  • Checkpoints are replicated asynchronously to the backup VM
Remus Implementation: Snapshots/Checkpoints

- Execution is divided into epochs. At the end of an epoch, do:
  a. Pause VM (End epoch i)
  b. Copy changed state into buffer
  c. Unpause VM (Start epoch i+1)
  d. Send buffer to backup
  e. Wait for ACK from backup to primary
  f. Release all output from epoch i

Figure 1: Speculative execution and asynchronous replication in Remus.
Discussion: Container Migration and Replication

• Containers are essentially processes (process migration/replication?)

• Can we use the same pre-copy memory migration mechanism?

• What about network and storage?

• What about replication?

• Migration and replication policies (container orchestration)?

• Any new opportunities?

• Any new challenges?