Remus: High Availability via Asynchronous Virtual Machine Replication

Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield

Department of Computer Science,
The University of British Columbia

Presenter: Lei Xia
Feb. 25 2009
Outline

- Motivation
- Approach
- Design and Implementation
- Evaluation
- Conclusion and Future work
Motivation

- It’s hard and expensive to design highly available system to survive hardware failure
  - Using redundant component, special-purpose hardware.
  - Reengineering software to include complicated recovery logic.
Motivation

The goal is to provide high availability system, and it’s:

- Generality
  - Regardless of applications and hardware
- transparency
  - Without modification of OS and App.
- Seamless hardware failure recovery
  - No externally visible state lost in case of single-host failure
  - Failure recovery should be fast
Approach

- **VM-based whole system replication**
  - Frequently checkpoint whole Virtual Machine state.
  - Protected VM and Backup VM is located in different Physical host.

- **Speculative execution**
  - We buffer state to *synchronous* backup later, and continue execution ahead of *synchronous* point.

- **Asynchronous replication**
  - Buffering output at the primary server allows replication to be performed *asynchronously*.
  - Primary VM execution is overlap state transmission.
Speculative execution and replication in Remus
Design and Implementation

Failure Model

- The fail-stop failure of any single host is tolerable.
- If both host fail, protected system’s data will be left in a crash-consistent state.
- No output will be made externally visible until the associated system state has been committed.
Design and Implementation

- Remus implementation is based on:
  - Xen’s support for live migration to provide fine-grained checkpoints.
  - Two host machines is connected over redundant gigabit Ethernet connections.
- The virtual machine does not actually execute on the backup host until a failure occurs.
Remus: Architecture

Figure 2: Remus: High-Level Architecture
Pipelined checkpoint

- Checkpointing runs in high frequency.
  - **Step 1:** Pause the running VM and copy any changed state into a buffer.
  - **Step 2:** With state changes preserved in a buffer, VM is unpaused and speculative execution resumes.
  - **Step 3:** Buffered state is transmitted to the backup host.
  - **Step 4:** When complete state has been received, acknowledge to the primary.
  - **Step 5:** Finally, buffered network output is released.
Checkpoint Machine State

- CPU & memory state
  - Checkpointing is implemented above Xen’s existing code for performing **live migration**.

- live migration
  - Technique by which a VM is relocated to another physical host with slight interruption.
Xen’s live migration

- Stage 1. Memory is copied to the new location **while** the VM continues to run at the old location.
- Stage 2. During migration, writes to memory are intercepted, and dirty pages are copied to the new location in rounds.
- Stage 3. After a specified number of intervals, the guest is suspended and the remaining dirty page and CPU state is copied out. (final round, stop-and-copy)
- By hardware MMU, page protection is used to trap dirty page.
- Actually, Remus implements checkpointing as repeated executions of the final round of live migration.
Modification to Xen Live Migration

- Goal: 1) performance; 2) ensure a consistent image is always available at the remote location.

- Migration Enhancements
  - Checkpoints support
  - Asynchronous transmission
  - Guest modification
Network buffering

- Most networks cannot provide reliable data delivery.
  - Therefore, network applications use reliable protocols to deal with packet loss or duplication.
- This simplifies the network buffering problem: transmitted packets do not require replication.
Network buffering (cont’d)

❖ To ensure packet transition atomic and checkpoint consistency:

➢ Outbound packets generated since the previous checkpoint are queued. And

➢ Released until that checkpoint has been acknowledged by the backup site.

➢ Inbound packets are delivered to host directly
Disk Buffering

- Requirements
  - All writes to disk in VM is configured to write through.
  - Recovery from single host failure
  - Preserve crash-consistent when both hosts fail.
- On-disk state don’t change until the entire checkpoint has been received
Disk Buffering

- Maintaining complete mirror of active VM’s disk on the backup host
  - Writes to storage are tracked and checkpointed
- All writes to active VM’s disk are write throughg
  - Immediately applied to primary disk
  - Asynchronously mirrored to backup’s memory buffer
  - No on-disk state changed until the entire checkpoint has been received
Disk Buffering

Figure 4: Disk write buffering in Remus.
Detecting Failure

- Use a simple failure detector directly integrated in the checkpointing stream.
- Timeout event represent the host’s failure.
  - A timeout of the backup responding to commit requests.
  - A timeout of new checkpoints being transmitted from the primary.
Evaluation

- **Correctness**
  - Kernel compiling with X11 client
  - 25 ms checkpoint
  - Every failure point, 1s delay on network, no inconsistency in backup disk image
Evaluation

Figure 5: Checkpoint time relative to pages dirtied.
Evaluation (cont’d)

Figure 6: Kernel build time by checkpoint frequency.

Figure 7: SPECweb scores by checkpoint frequency (native score: 305)
Evaluation (cont’d)

Figure 8: The effect of network delay on SPECweb performance.
Evaluation (cont’d)

Figure 9: The effect of disk replication on Postmark performance.
Conclusion

- A VM-based software method to provide high availability to survive hardware failure, with low cost and transparency.
Limitations

- Outbound packet latency, lower network throughput
- Performance