Resource Management With Virtualization

Agenda

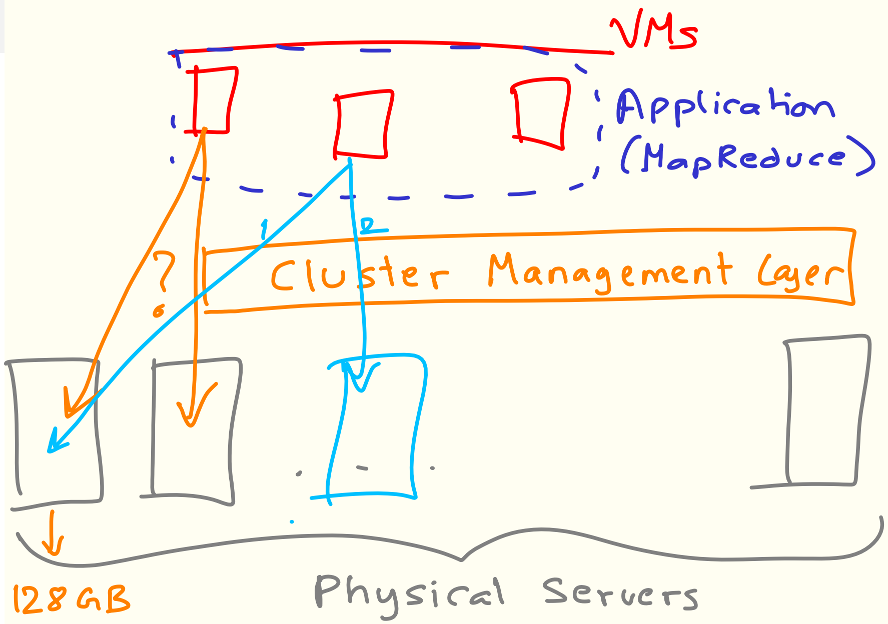
- Cluster-level resource management
- VM Resource Overcommitment

VM Sizes

Hypervisor allocates all VMs with many resources:

1. CPU cycles (i.e., bandwidth) , or CPU cores
2. Physical memory
3. Disk bandwidth — I/O operations/second (iops)
4. Virtual disk size
5. Network bandwidth — 8 Gbps
6. More recently: Special purpose accelerators (GPUs, FPGAs, ASICs) TPUs

Common to express resource allocations in form of resource vectors.



VM: {(0,128]

Virtualization For Resource Allocation

- Virtualization makes fine-grained resource allocation easy

1.5 vCPUs

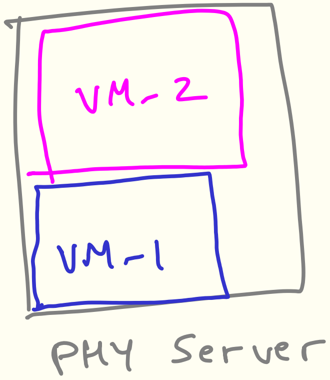
- VMs serve as units of allocation
- Resource management layer (i.e., OS or hypervisor) can set resource limits on the VM
- Resource limits can often by dynamically changed (e.g., reduce CPU allocation to 2 cores from 4)

"Elastic" allocation

Resource Allocation In Clusters

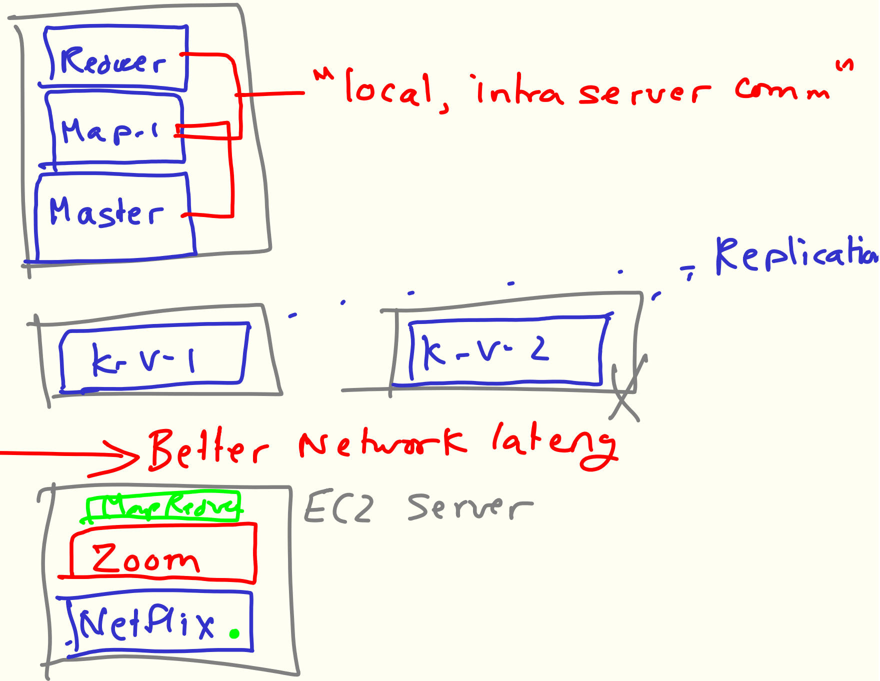
- Clusters consist of large numbers of servers $(10^2 - 10^6)$
- Resources can be allocated from **multiple** servers
- Resources allocated as VMs on individual servers
- Allocation decisions made by cluster management software
    - OpenStack, VMWare for VMs
    - Kubernetes, Mesos, Docker swarm for containers
    - Slurm, Torque for HPC...

Scale, Efficiency, ...



High-Level Resource Allocation Flow

1. Applications/Users submit resource requirements ($\mathbf{R}$)
   - Total number of resources (CPU cores, memory, I/O bandwidth), or
   - Size of VM × number of VMs
2. Each server has a hardware capacity (e.g., 48 cores, 512 GB memory) ($\mathbf{C}$)
3. Cluster manager finds free resources on servers to satisfy allocation request
4. In practice, many other allocation constraints:
   - Application quotas: does user have enough "credits"
   - Job start/end deadlines → HPC, Scientific Computing, Batch
   - Affinity: VMs should be running on same/nearby servers → Better Network lateng
   - Anti-affinity: VMs on different servers for fault tolerance
   - Co-location: Applications should not be running on servers with another application

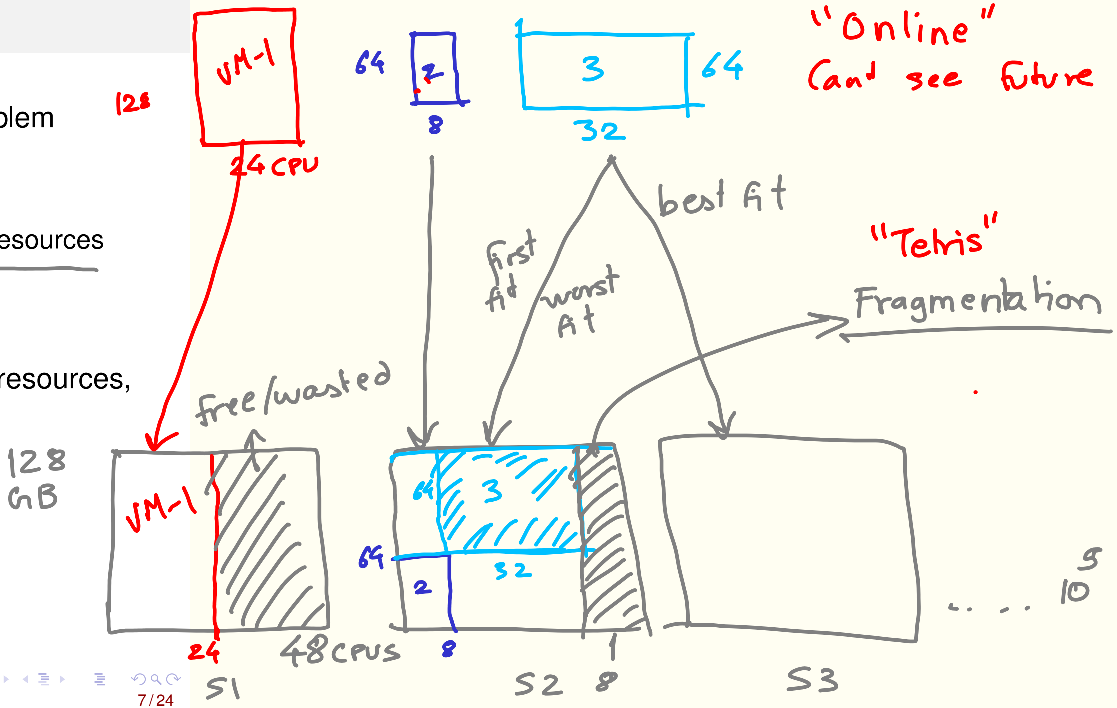


Resource Allocation Policies

- At a high level, resource allocation is a bin-packing problem
- Also called the “placement” problem
- Which servers to place the VMs on?
  - Best fit: Allocate resources from server with most free resources available
  - Worst fit: Server with least free resources
  - First fit: Sort by server-id
- However, this is a *multi dimensional* packing problem : resources, r=(CPU, mem, disk, network) ≡ VM size

· Offline, single dim bin packing : NP Complete

· Easy approximate algorithm (Greedy First Fit) : 11/9



"Online"
Can't see future

"Tetris"
Fragmentation

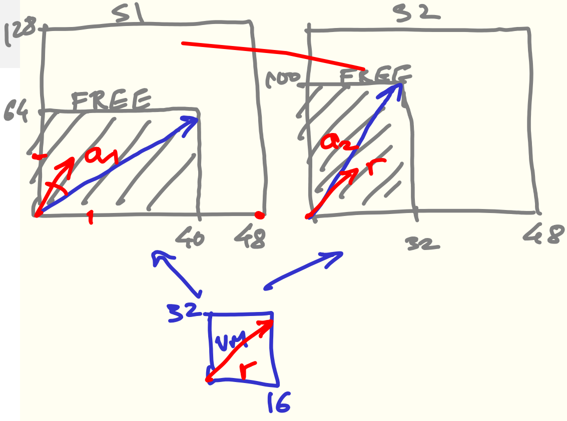
Multi-dimensional Packing

- Use cosine similarity between resource requirement and availability vectors: fitness = $\frac{\mathbf{r}\cdot\mathbf{a}}{|\mathbf{r}||\mathbf{a}|}$ $|a| = \sqrt{a_{CPU}^2 + a_{Mem}^2}$
- $\mathbf{a}$ is the resource availability on the server
- $\mathbf{a}$ = Server Capacity $-\sum$ VM sizes

Other heuristics also possible:

Res type: CPU, Mem

- L2-norm-diff : $\sum (r_i - a_i)^2$
- L2-norm-raito: $\sum \frac{r_i}{a_i}$
- First-fit-decreasing prod: $\prod r_i$
- FFD-sum: $\sum r_i$



bestfit: most amt of free resources.

Q: How to quantify free space in multiple dimensions?

① ||Free||

② $\sum_{i \in d} f_i$

③ Prefer most saturated

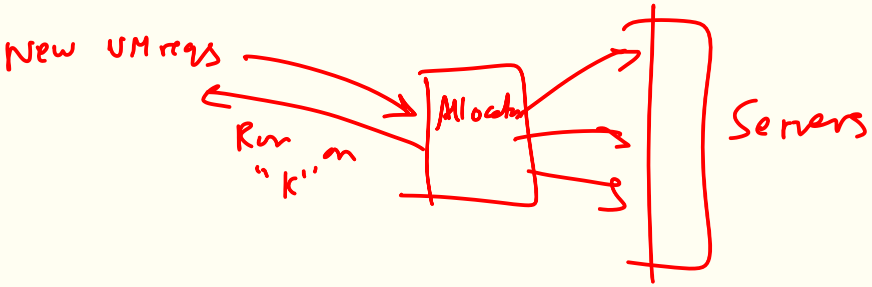
free = mem, use cpu to break ties

⋮

Centralized Resource Allocation

- Cluster manager runs on a single server
- Resource allocation state is centralized
- Set of available servers, resources on each server, map of applications to servers, ...
- If a new application wants resources:
  1. Find best allocation according to placement policy
  2. Update local state (server resource map)
  3. Allocate resources in form of containers/VMs..
- All the advantages and drawbacks of a centralized approach
- Used by Kubernetes, Slurm, OpenStack, VMWare,....

Containers MPI VMs

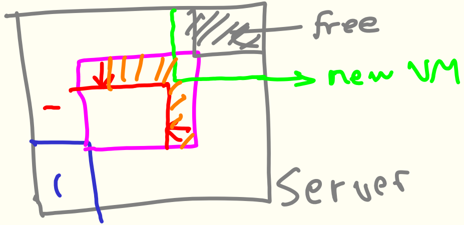


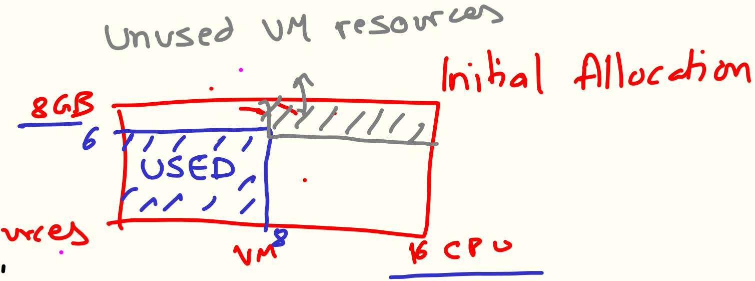
VM Overcommitment

- Hypervisors can also *overcommit* resources allocated to VMs
- VMs are "committed" C resources, but can only effectively use c, where $c < C$.
- VM's "true" resource allocation effectively reduced
- This process is called *resource reclamation*
- **Useful to "pack" more VMs onto a server**

Overcommitment Types

- **Transparent:** The guest OS/applications cannot "tell" that resources have been reclaimed by the hypervisor.
- **Explicit:** Guest OS has knowledge of the reclamation, and may even cooperate in the overcommitment process. — Return unused resources

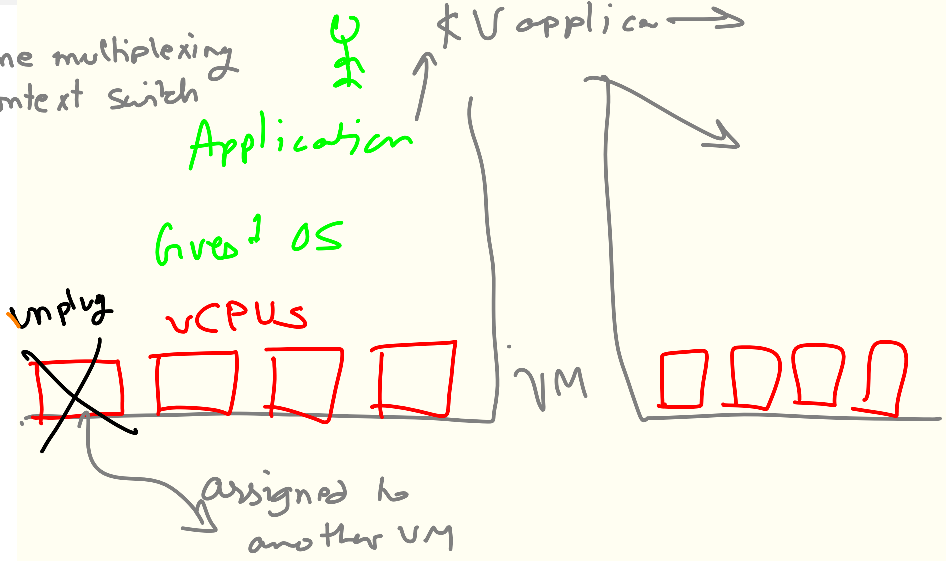




CPU Overcommitment

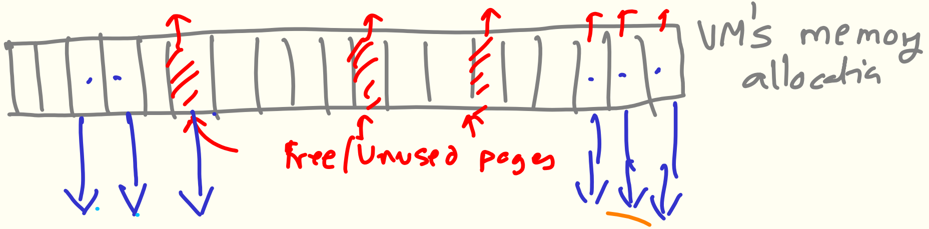
- Hypervisors schedule vCPUs to run (just like the OS schedules processes)
- Hypervisors can thus reduce a VMs CPU allocation by scheduling its vCPUs less often
- This is **transparent.** Guest OS/application have no direct way of knowing, and do not need to be modified.
- **Explicit mechanism:** vCPU hot-unplug
- With hot-unplug, a vCPU can be “removed” from the VM.
- Guest OS and applications see a reduction in total amount of vCPUs available.

Time multiplexing
Context switch



Memory Overcommitment

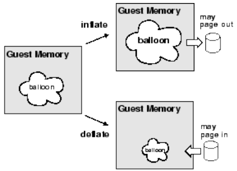
- **Transparent:** Hypervisor swaps out the VM's memory pages.
- **Explicit:** Some amount of memory is hot unplugged.
- Hot-unplugging of memory is...complicated
- Guest OS must cooperate and find and return unused pages.
- Another popular explicit reclamation technique is **ballooning.**



Swap to disk,
& allocate the phy pages to another VM.

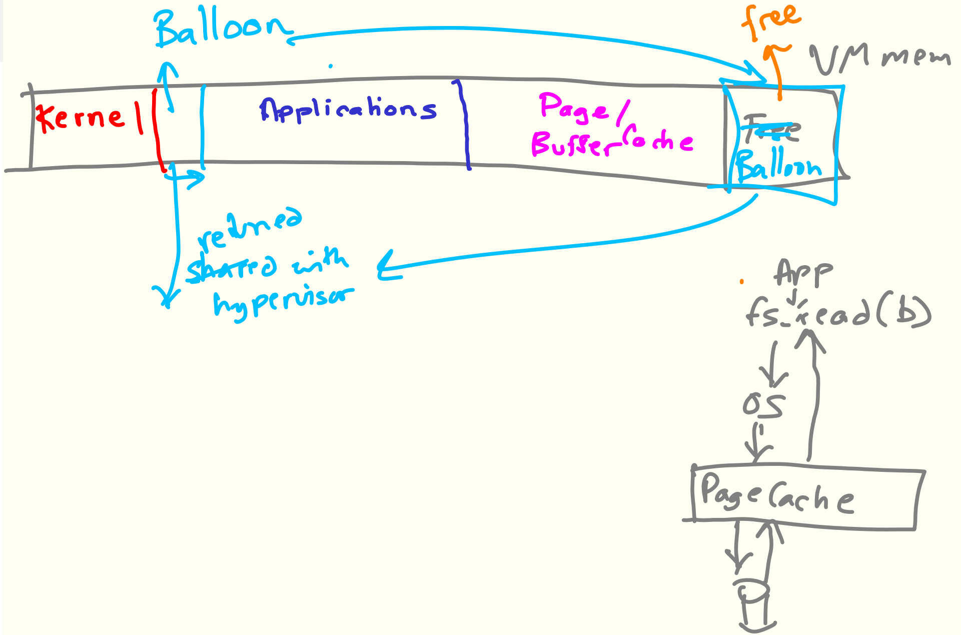
Memory Ballooning

- Ballooning pre-dates hot-unplug, and was required when guest OSes did not support hot-unplug.
- Guest OS is installed with a balloon driver, which allocates large amounts of memory
- The memory requested by the balloon is given to the hypervisor, so that it can allocate it to other VMs.



Reading

"Memory Resource Management in VMware ESX Server." Carl A. Waldspurger.



Transparent vs. Explicit Overcommitment Tradeoffs

- Transparent techniques may hurt VM performance more — "reclaim wrong resources"
- If Guest OS/application is notified about it being shrunk, it can make better resource allocation decisions
- Example: Most memory is used for disk caching (page cache)
- Guest OS can discard some cached items when balloon expands
- Hypervisor level Transparent Overcommitment is "blind" and may move "wrong" pages to swap.

More memory Overcommitment

Main problem with overcommitment:

- Overcommitment reduces VM performance!
- **Is there a way to overcommit without affecting VM performance?**

Overcommitment is not so bad!

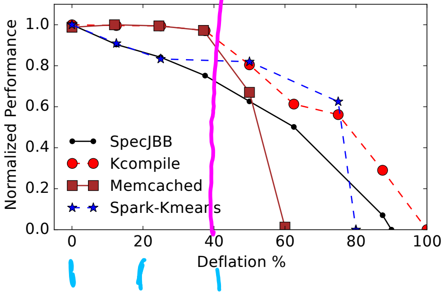
- In many cases, resources can be overcommited safely without much performance penalty.
- Mainly because reclaiming resources *not used* by the VM should not affect performance
- Luckily, most applications use a small fraction of VM resources
- VMs are typically *over-provisioned* by customers

VM utilization public clouds 20-50%

= very low

Application Performance With Overcommitment

- Performance of application with overcommitment depends on overprovisioning and application characteristics.
- Usually, resources can be reclaimed to a large extent without the proportional performance reduction
- “Utility curves” have this typical shape:

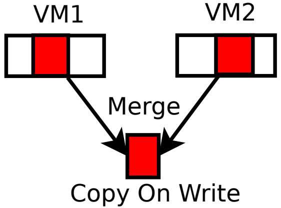


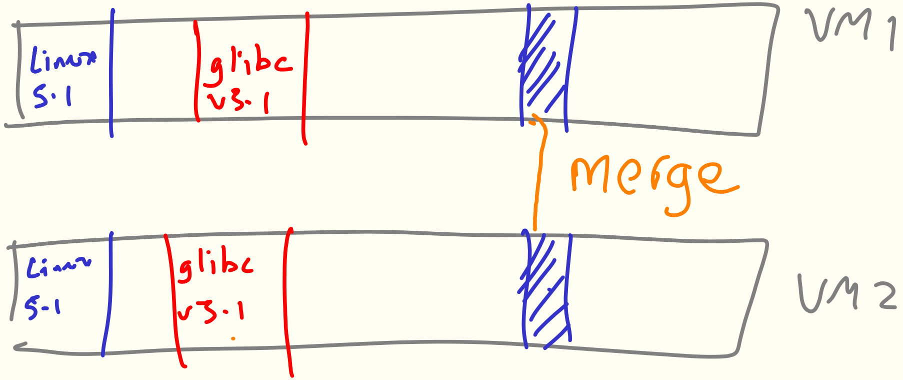
SpecJBB: Interactive app benchmark

Reduction in performance NOT Linear

Memory Overcommitment with Page Deduplication

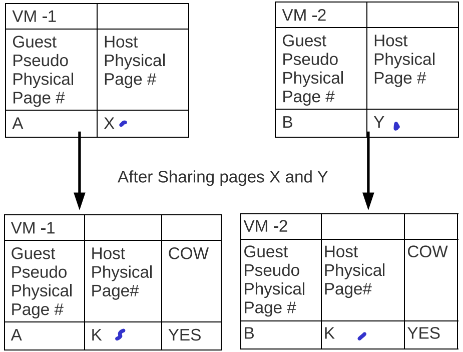
- Many VMs run the same OS (Linux), libraries (glibc, python, . . .), and software (apache, memcached, . . .)
- Guest OS code, libraries, and application code occupies significant amount of VM memory





Page Deduplication

1. Hypervisor constantly scans and finds duplicate pages → large hash table of page checksums
2. Duplicate pages → Exactly same content
3. Same libraries, application binaries, data, etc.
4. Duplicate pages are *merged* by Hypervisor
5. Merged page is marked copy-on-write for safety



VM -1	
Guest Pseudo Physical Page #	Host Physical Page #
A	X

VM -2	
Guest Pseudo Physical Page #	Host Physical Page #
B	Y

After Sharing pages X and Y

VM -1		
Guest Pseudo Physical Page #	Host Physical Page#	COW
A	K	YES

VM -2		
Guest Pseudo Physical Page #	Host Physical Page#	COW
B	K	YES

PHY 2 Pages

↓

1 PHY Page + 1 Free Page

1 VM 2 writes to K

2. CoW fault → hypervisor

3. Copy K → ~~Y~~ K2

PT :

A → K . no CoW

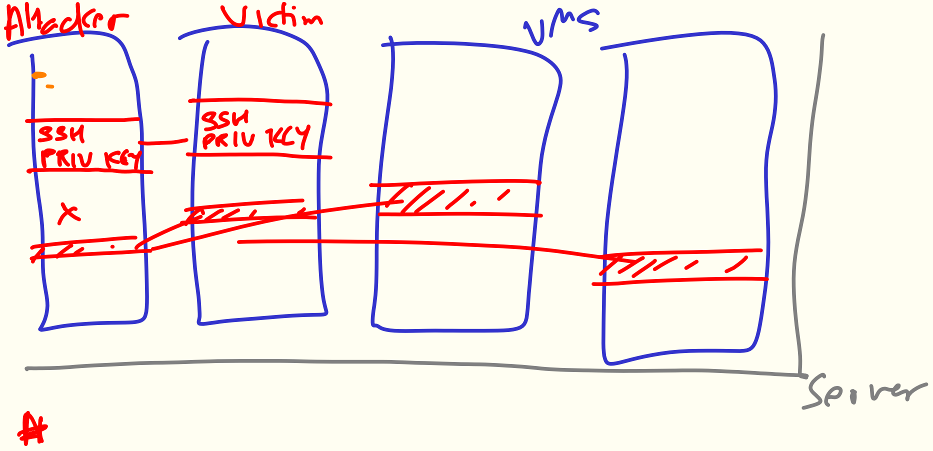
2. B → K2 . no CoW

More On Page Deduplication

- Effective VM memory footprint reduced *without* actually reducing its memory allocation
- Completely transparent to VM, even wrt performance!

Downsides?

- **Timing side channels!**
- Attacker VM can find out what code version a victim VM is running
- 1 Generate "random" pages. Assume Aels shared
- 2 Write to them after a while
- If write operation takes slightly more time, it is because the page was marked copy-on-write, and the Hypervisor had to make a copy.
- Also maybe steal encryption keys.



Cluster Load-balancing with Migration

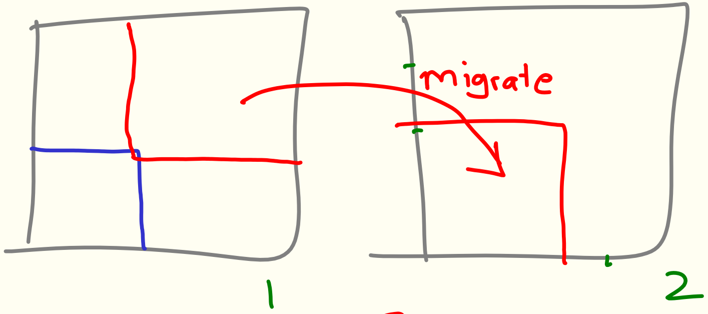
- Due to Overcommitment on a server or otherwise, VM may face performance degradation
- Key idea: Live-migrate VM to a less loaded server

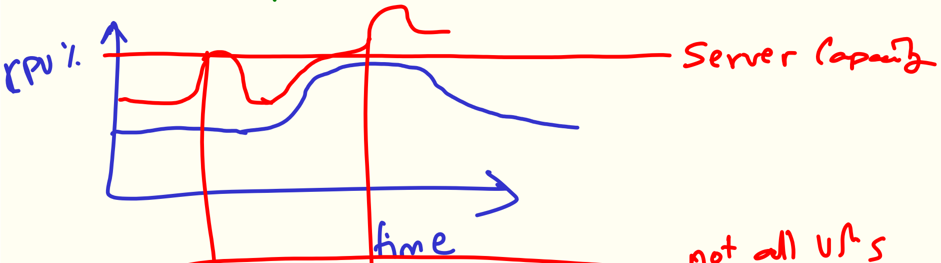
Black and gray box overload detection

- Black-box: Look at VM-level metrics that hypervisor can access
- VM CPU utilization, I/O rate, etc.
- Gray-box: Application and OS level metrics
- Respose time, memory usage inside VM, etc.

Reference

“Black-box and Gray-box Strategies for Virtual Machine Migration”, T. Wood et. al.





Statistical Multiplexing : not all VMs are going to peak at once

on "average", the VM util is low.

Virtualization for fault-tolerance

"Exact" same state at "all" times

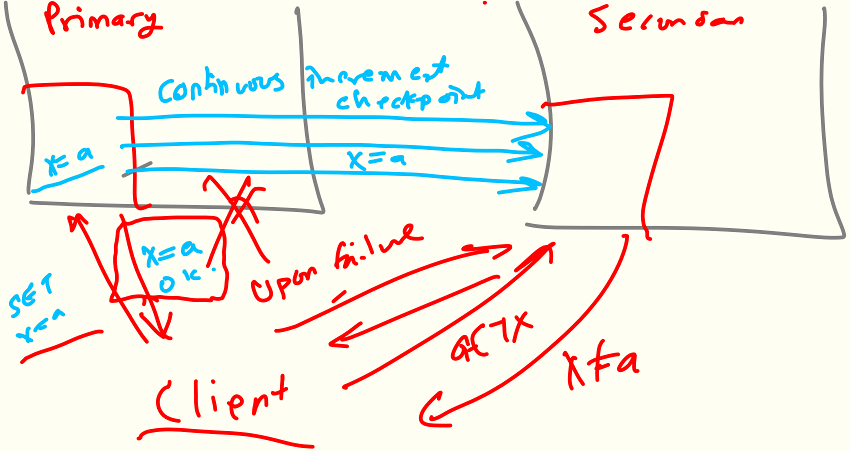
- What if the server hosting a VM fails?!
- Key idea: Primary-secondary replication
- Run two identical VMs. If one fails, the other can seamlessly take over

Remus

- Checkpoint and migrate VM memory state to secondary server
- Very frequent Checkpointing: every ~ 100 milliseconds
- Key trick: Buffer all outgoing network packets until memory is synced

Reference

Remus . Warfield et. al.



VM-fork

- Analgous to process fork
- Want to clone a VM and launch it on another server
- Both parent and child VMs continue running
- Useful for increasing parallelism and horizontally scaling

SnowFlock

- Copy memory state using post-copy migration
- Child pages are copied on first access, over the network.
- All parent VM pages are marked copy on write

Reference

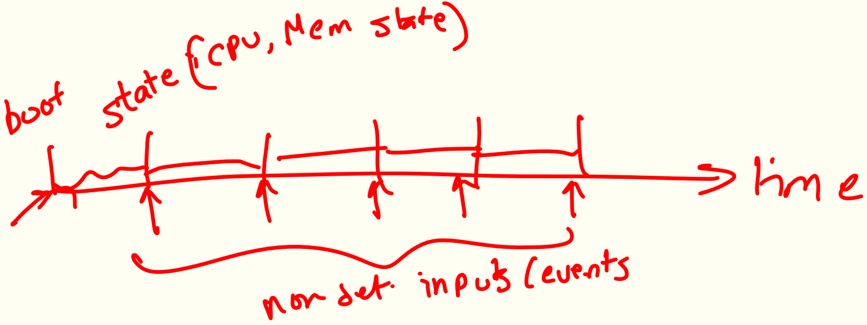
SnowFlock

Record-replay

- Useful for debugging
- Record only non-deterministic events
- Replay them at exactly the time they occured at.

random #'s
interrupts
external inputs

- File read
- User Kb/Mouse Input
- Network

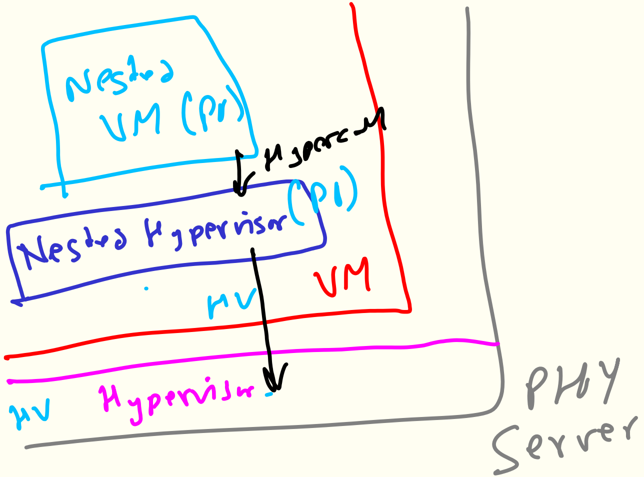


Nested Virtualization

- Run a VM inside a VM!
- XenBlanket: PV VM inside a HVM VM
- Hypercalls are proxied

Paravirtualize

Hardware CPU, Page Table exten



Nested VM (PV)
Hypercall
Nested Hypervisor (PV)
HV
VM
HV
Hypervisor
PHY Server