KVM Memory Optimizations

Improving Memory-management with KSM

Prateek
Advisor: Puru

September 24, 2011

High-Level Goals

- Increase number of VMs without degrading performance.
- Improve how guests (and hosts) utilize physical memory.
- Why Memory : Most constrained and non-renewable resource.
- Allocated to guests at their boot time.
- Memory overcommit is one of the key drivers of virtualized hosting
- *Everyone wants 8GB, even if they are using a few MB*
- Dynamic memory management in VMMs using page-sharing.
- **KSM = Kernel Samepage Merging**

Page-Sharing

Assume pages X and Y have same content.

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

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

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

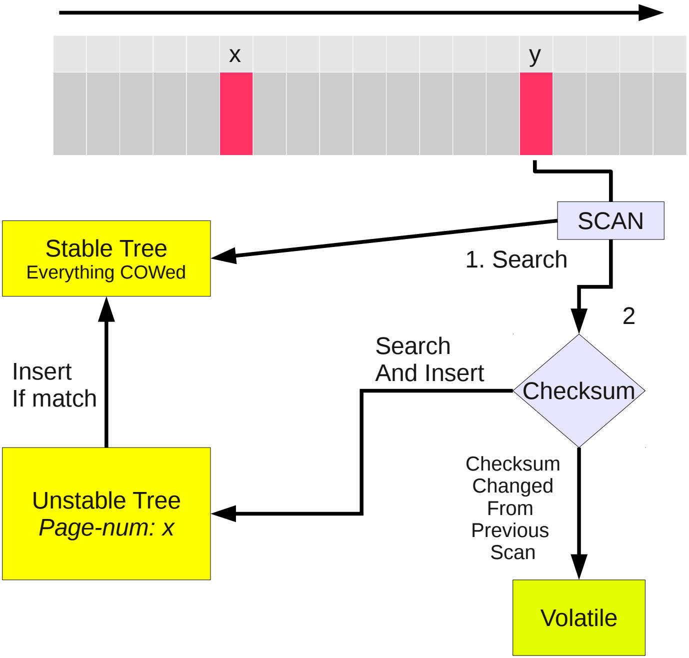
VM -2		
Guest Pseudo Physical Page#	Host Physical Page#	COW
B	K	YES

KSM & Page-Sharing

- KSM implements *Content Based Page Sharing* in linux.
- Multiple pages with the same content are merged into one.
- Detects similarity among ever-changing objects(pages)
- Brute-force search at regular intervals : scanning.
- Different implementations : VMWare ESX, Difference Engine, Satori, KSM.

This talk: Describe 3 modifications to KSM

KSI



x
y
SCAN
Stable Tree
Everything COWed
1. Search
2
Search
And Insert
Checksum
Insert
If match
Unstable Tree
Page-num: x
Checksum
Changed
From
Previous
Scan
Volatile

KSM

The Good:

- Very general and non-disruptive solution - meant for page sharing in arbitrary memory areas.
- Any `malloc`'ed area is shareable (via `VM_MERGEABLE`)
- Very few heuristics used.

The Bad:

- Significant overhead due to continuous scan+compare
- 10-20 % CPU utilization in most cases.

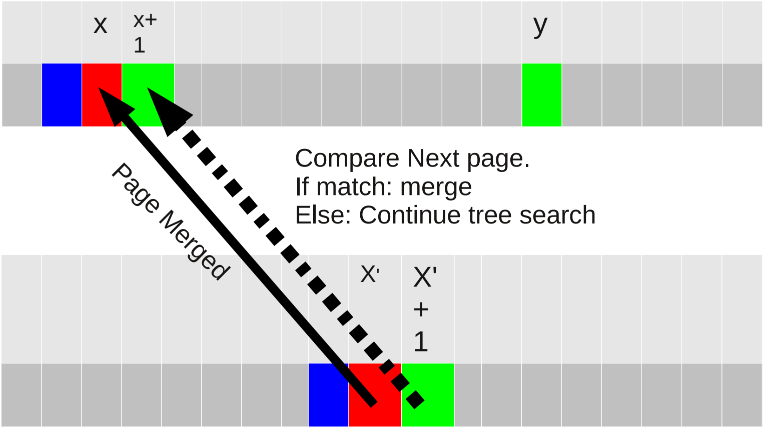
The Ugly :

- Design constrained by a patent minefield.

Lookahead

- Shared pages in VMs are contiguous (seen using `ftrace`)
- Lots of shared pages are file-backed. (guest `kpageflags`)
- Modify KSM search to account for this spatial locality
- Peek at next page before doing the tree-search
- Assuming consecutive shared pages occour with probability p , Reduce search costs from log(u) to (1-p)log(u) + (p)1.

Lookahead



x
x+
1
y
Page Merged
Compare Next page.
If match: merge
Else: Continue tree search
X'
X'
+
1

Lookahead

- Lookahead optimization has no overhead in the worst case.
- Shared pages **increase** because KSM can scan lot more pages per CPU cycle.
- Ideal situations: sharing of large files.
- Desktop environments are best. (X11 fonts, programs, etc).
- Sub-optimal situations: Lots of sharing, but fragmented. (Kernel compile)

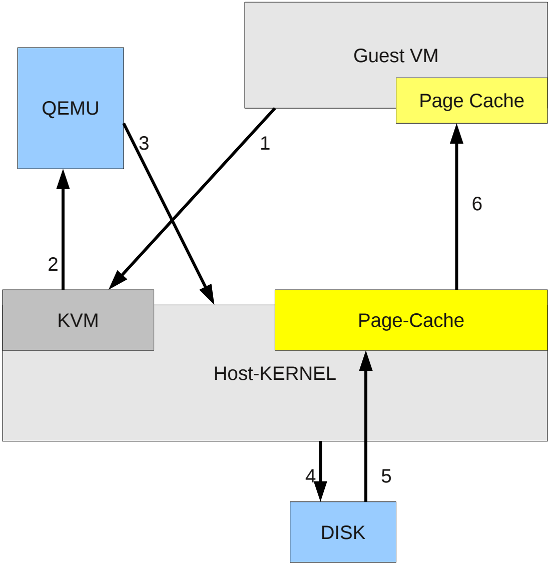
Lookahead results

Workload (2VMs)	Avg. Shared Pages - Vanilla	Avg. Shared Pages - lookahead	CPU-look	CPU-Vanilla
Boot up	8,000	11,000	12	12
Kernel Compile	26,000	30,000	22	19
Desktop VM use	31,000	62,000	16.8	14.6

Table: Lookahead performance

Average shared pages (over time) during the course of the workload.
CPU usage is also the average over time.

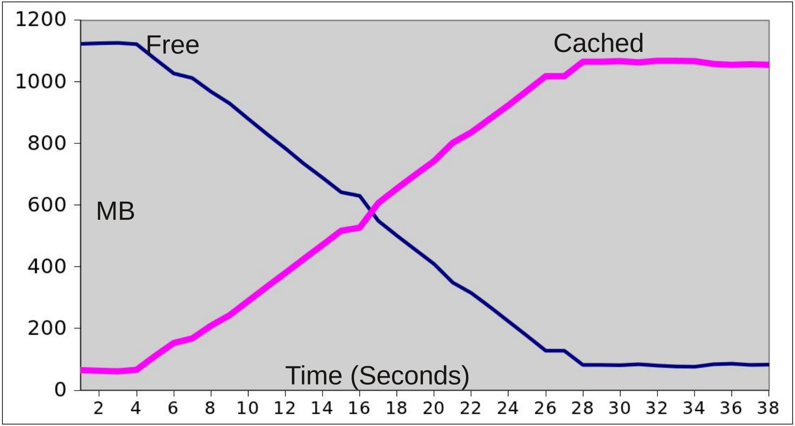
QEMU IO


Guest VM
Page Cache
QEMU
3
1
6
2
KVM
Page-Cache
Host-KERNEL
4
5
DISK

Exclusive Caching

- All guest disk IO goes through host page-cache.
- Double-caching : Same block is present in both host and guest caches.
- Ideally a block should be present in either of the caches.
- Exclusive caches are known to provide better cache utilization. [See: Geiger, gill, mycacheyours].
- Mounting virtual disks as `O_DIRECT` adds too much penalty.
- No solution to this for KVM-like systems exists . . .

Host memory cache pollution with yes workload



Double caching problem

Workload (2VMs)	Avg. Shared Pages	Total Pages dropped	Avg Cache saved	CPU-ksm	CPU-exc
Kernel Compile(2 GB)	75,000		512M - 260M	14	16
Desktop VMs	62,000	162,000	400M- 219M	18.8	14.6

Table: Exclusive Cache benefits. Significant reduction in host page cache size. Increased CPU usage due to cache scrubbing.

Exclusive Caching

- Wasting memory by caching 2 copies clearly wasteful
- No existing solution for virtual setups
- Use KSM!
- Drop a page from host page-cache if it's already present in guest.
- Luckily for us, KSM builds a nice search tree of all guest pages!
- Scan at the end of every KSM pass, comparing host page-cache pages with unstable,stable tree.
- If match found, drop page from host cache

Evaluation of Exclusive Cache

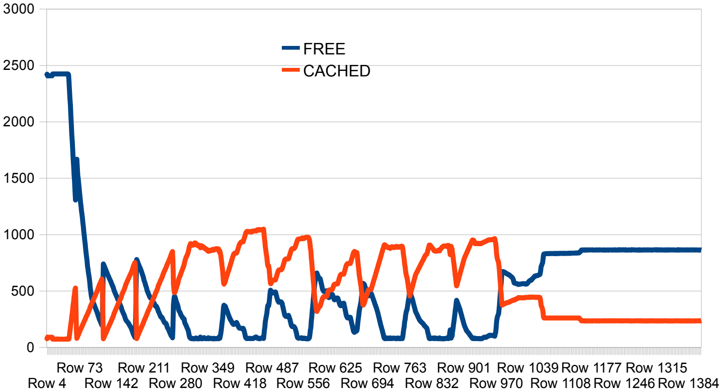
Test	Plain	With Excl Cache
Write(char)	24,000 K/s	32,000 K/s
Read(char)	27,000 K/s	27,500 K/s
Read(block)	53,750 K/s	47,700 K/s
Write(block)	22,500 K/s	23,800 K/s

Table: Exclusive cache impact on Bonnie++ benchmark. Executed on 2 VMs.

- The caches are not scrubbed often enough in this short-lived benchmark
- Nevertheless significant performance gains

Cache scrubbing in action

Free and Cached memory (in MB) for a sample Bonnie workload. Obsserve the page-dropping at regular intervals.



Kernel-hacking timeline

KSM page-flags	1 Month (!)
Ftrace instrumentation	2 days
Lookahead implementation	3 weeks
Exclusive Cache implementation	2 hours

Table: Time spent

- Currently on kernel version **195** (number of times compiled)
- Compile → Reboot → Debug cycle is exhausting.
- Tricks : QEMU gdb mode , kexec , ksplice.
- The kernel infrastructure is *amazing* : `Lockdep` (detect deadlocks!) , `kmemcheck` , `debug_info` ...

TODO

- More experiments, with different workloads.
- Need low-intensity, high-sharing benchmarks.
- Zipf workload comparison with truly exclusive vs. KSM-exclusive vs. Drop-all.
- Implement WSS estimator using KSM. (detect evictions).
- More KSM improvements to be tested .

Page Sharing by Flags

- Hypothesis: Sharing page-cache pages is bad.
- Page-cache pages are overwritten frequently anyway.
- Page-cache size is 50% of available memory , so significant KSM savings.
- This hypothesis turns out to be *wrong*

Implementation

- Guest writes its page-flags into a memory hole.
- Host (KSM) needs to access statically defined address in the guest. HOW?
- Kernel doesnt seem to have a mechanism to provide memory by physical addresses
- Currently : Create memory-hole at boot-time and write to it (using `ioremap`)