Difference Engine: Harnessing Memory Redundancy in Virtual Machines

Diwaker Gupta, Sangmin Lee,Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat, OSDI08

Presenter: Orna Agmon Ben-Yehuda

Department of Computer Science
Technion — Israel Institute of Technology

Advanced Topics in Computer Systems, Computer Science Seminar 5 (236805), Spring 2010

Why Share Memory?

- Server consolidation saves money and energy.
- Memory is a key bottleneck for VM consolidation.
- Sharing enables memory over-commitment.

Memory Over-Commit Mechanisms for Virtualization

1. Ballooning - well established, drivers widely available
2. **Page Sharing**
   - **Collaborative** (e.g. Satori: Enlightened Page Sharing, by Milos et al.) - requires guest modification (paravirtualization) or **by hypervisor only** (unmodified guest OS - full virtualization)?
   - **Content based** or **by tracking changes** (faster, requires guest changes, e.g., Satori)
   - **Whole pages** (Waldspurger of VMware, OSDI'02) or **sub-pages**?
3. Paging - orthodox but slow

Memory Over-Commit Mechanisms for Virtualization

1. Ballooning - well established, drivers widely available
2. **Page Sharing**
   - Collaborative (e.g. Satori: Enlightened Page Sharing, by Milos et al.) - requires guest modification (paravirtualization) or **by hypervisor only** (unmodified guest OS - full virtualization)?
   - **Content based** or **by tracking changes** (faster, requires guest changes, e.g., Satori)
   - **Whole pages** (Waldspurger of VMware, OSDI'02) or **sub-pages**?
3. Paging - orthodox but slow

Memory Over-Commit Mechanisms for Virtualization

1. Ballooning - well established, drivers widely available
2. **Page Sharing**
   - Collaborative (e.g. Satori: Enlightened Page Sharing, by Milos et al.) - requires guest modification (paravirtualization) or **by hypervisor only** (unmodified guest OS - full virtualization)?
   - **Content based** or by tracking changes (faster, requires guest changes, e.g., Satori)
   - **Whole pages** (Waldspurger of VMware, OSDI'02) or **sub-pages**?
3. Paging - orthodox but slow

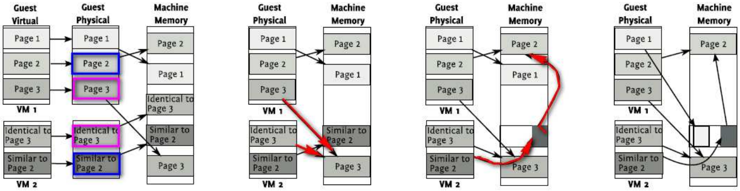
Memory Over-Commit Mechanisms for Virtualization

1. Ballooning - well established, drivers widely available
2. **Page Sharing**
   - Collaborative (e.g. Satori: Enlightened Page Sharing, by Milos et al.) - requires guest modification (paravirtualization) or **by hypervisor only** (unmodified guest OS - full virtualization)?
   - **Content based** or by tracking changes (faster, requires guest changes, e.g., Satori)
   - Whole pages (Waldspurger of VMware, OSDI'02) or **sub-pages**?
3. Paging - orthodox but slow

Outline

Cascade of Mechanisms

Initial ⇒ Share, **Patch**, Compress, Page



(a) Initial (b) Page sharing (c) Patching (d) Compression

Sharing

- Identify: Hash collision + verification
- Share: directing guest pages to the same physical page, read only
- Break: Copy On Write (COW)
- Clean: when 0 references

Mixed Real-World Workloads

Each VM with 512 MB. Stressing memory. Following VMmark (VMware), VMbench (Moeller, PhD thesis)

1. Mixed-1
   1. Windows, running RUBiS (e-commerce: Apache+MySQL)
   2. Debian, compiling Linux kernel
   3. Slackware , compiling Vim, then running lmbench (memory, network, filesystem, signals....)
2. Mixed-2
   1. Windows, Apache with 32K static pages requested by external httperf
   2. Debian, Sysbench (db) with 10 threads creating 100K requests
   3. Slackware, dbench (filesystem) with 10 clients for 6 minutes, then IOZone (filesystem)

Potential Estimate for Patching and Sharing

- Ran Mixed-1
- Suspended the VM after completing the benchmark
- Took memory snapshot
- Computed patches with XDelta (FOSS binary diff)
- Patch limit: 2K (half a page), average patch: 1K
- Zero pages appear less when VMs get less memory, *when scrubbing is used less, when Linux caches more files*

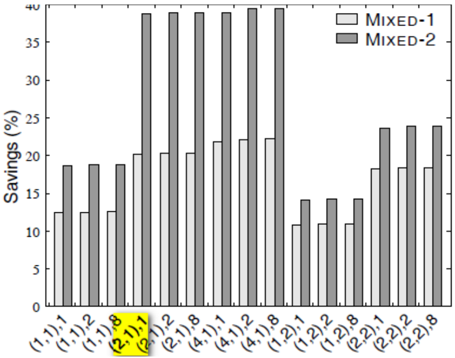
Pages	Initial	After Sharing	After Patching
Unique	191,646	191,646	
Sharable (non-zero)	52,436	3,577	
Zero	149,038	1	
Total	393,120	195,224	88,422
Reference		50,727	50,727
Patchable		144,497	37,695

HashSimilarityDetector(k,s),c

- Hash $k \cdot s$ randomly chosen 64-byte block locations on the page
- Group to k groups, each group is an index in the hash table
- HashSimilarityDetector(2,1): two keys for each page, two indexations (a candidate needs to match only one)
- c: number of different pages stored for each key (choose the best patch among the stored pages)
- Smaller k,s,c $\Rightarrow$ less resources used

Savings from Patching Only as Function of k,s,c

$$mem\ savings = 100\% \cdot \left(1 - \frac{used}{allocated\ to\ VMs}\right)$$



Chosen: 2 hash keys of single locations. 1 stored page. 18-bit hashes (32 bit hashes yields: 25% instead of 20% for mixed-1).

Compression

- When:
  - Compression ratio is high enough.
  - Page is infrequently accessed - “Not Recently Used” (NRU).
  - Page is unique.
- Compressed page is invalidated, so the hypervisor knows when to decompress it.
- Pluggable: currently supports
  - LZO (Lempel-Ziv, very fast decompression, trade-off between compression speed and quality)
  - WKdm (fast encoding)
- Decompressed page remains open in memory until considered for compression again.

Paging

- Involves disk I/O - slow
- Extends beyond physical memory
- Candidates — NRU
- Swapped out pages cannot be shared or referenced for patching
- Safety net

Changes to Xen

- 14.5K lines added + 19K lines for existing libraries
- Changes mainly in guest physical to machine table, and in the shadow page tables
- Difference Engine (DE) not in effect during boot, only when shadow page tables are used
- Not touching Dom0 to avoid circularity
- ioemu (IO emulator, in Dom0) changed to map only several guest pages to Dom0.
- Block allocator - to efficiently manage storage of compressed and shared pages (consume less than one page).

Clock

Clock used to find NRU pages.

- Each invocation
  - Resets **R**ead, **M**odified bits
  - Scans a part of memory
  - Returns limited-size list of NRU pages
- Invocations at least 4 seconds apart
- Xen's shadow page tables code modified: setting those R/M bits in the guest physical to host physical map, based on the shadow page tables.

Clock Conditions: Policy/Mechanism Separation

recently=since last scan

1. C1 — Recently modified
2. C2 — Recently read only
3. C3 — Recently nothing
4. C4 — Nothing for several scans (needs 2 additional bits)

Clock Conditions: Policy/Mechanism Separation

Default Policy:

1. C1 — Recently modified
2. C2 — Recently read only
3. C3 — Recently nothing
4. C4 — Nothing for several scans

Clock Conditions: Policy/Mechanism Separation

Default Policy:

1. C1 — Recently modified — ignore
2. C2 — Recently read only
3. C3 — Recently nothing
4. C4 — Nothing for several scans

Clock Conditions: Policy/Mechanism Separation

Default Policy:

1. C1 — Recently modified — ignore
2. C2 — Recently read only — share/patch reference
3. C3 — Recently nothing
4. C4 — Nothing for several scans

Clock Conditions: Policy/Mechanism Separation

Default Policy:

1. C1 — Recently modified — ignore
2. C2 — Recently read only — share/patch reference
3. C3 — Recently nothing — share/patch
4. C4 — Nothing for several scans

Clock Conditions: Policy/Mechanism Separation

Default Policy:

1. C1 — Recently modified — ignore
2. C2 — Recently read only — share/patch reference
3. C3 — Recently nothing — share/patch
4. C4 — Nothing for several scans — anything

Clock Conditions: Policy/Mechanism Separation

Default Policy:

1. C1 — Recently modified — ignore
2. C2 — Recently read only — share/patch reference
3. C3 — Recently nothing — share/patch
4. C4 — Nothing for several scans — anything

Alternative policies:

- Consider all pages for anything - insignificant excess saving
- Compression before patching - slightly less savings, less performance overhead.

Sharing

- SuperFastHash
- Hash table needs to fit in Xen's limited memory (12M)
- *Constant* 5 passes, hashing 1/5 of the range each time : 1.76M page sharing hash table size

Patching

- *Similarity Hash Table is also stored in Xen itself, statically allocated, sized* 2^{18} *unsigned longs (1MB).*
- Clearing:
  - ⇒ At least after a full clock pass (=5× partial) - allows finding similarity between keys from different passes.
  - ⇐ Early clearing reduces stale data (pages changed after indexing).
  - = Similarity Hash Table cleared each full clock pass.
- Races:
  - Locking only when building patch and replacing page.
  - Other races only result in larger patches.

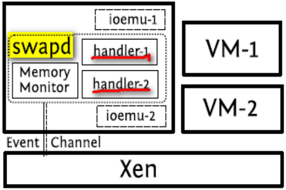
Compression

- Compression postponed till after all pages are checked for similarity (prevents patching)
- Condition C4 used to identify a complete cycle of page sharing checks.

Paging

Swapping implemented in Dom0, where Xen defers all I/O.

- A thread for each guest to handle swap-in requests
- A thread (memory_monitor) tracks system mem



swapd may initiate swap-out when:

1. Mem exceeds HIGH_WATERMARK (till LOW_WATERMARK achieved)
2. Xen notifies via event channel, e.g. for share break
3. Process requests via IPC (XenStore), e.g. for VM cloning

Paging - Cont.

Upon failure swapd continues silently:

- Full swap space
- No swap candidates

Implementation includes VM pausing. Actual swap file writing can happen asynchronously.

Paging: ioemu-swapd interaction

- Pages mapped by ioemu are ineligible for swapping out.
- ioemu mapped pages are swapped in before accessed, if needed.
- Race prevented by blocking ioemu when swapping-in (using shared memory).

Default Evaluation Setup

- 4 cores (dual processor, dual core 2.33 GHz Intel Xeon)
- Page size 4K
- *How much memory?*

Times of Individual Operations

Using micro benchmarks.

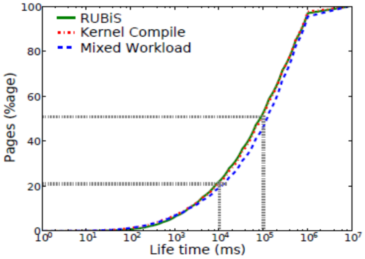
Function	Mean execution time (μs)
share_pages	6.2
cow_break	25.1
compress_page	29.7
uncompress	10.4
patch_page	338.1
unpatch	18.6
swap_out_page	48.9
swap_in_page	7151.6

Table 2: CPU overhead of different functions.

Swap-in may even take longer (swap file size, scheduling in Dom0,...)

Clock Performance - Lifetime of Patched/Compressed Pages

- A good clock should give high lifetimes to compressed/patched pages, which are costly to access.
- Performance of hetero workload close to homogeneous workload.
- *Good performance?*

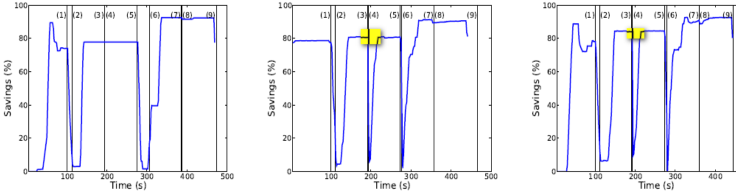


Isolated Mechanisms — Workload

- 4 steps:
  1. (1)-(2) Allocate pages (zero, random, identical, similar but not identical)
  2. (3)-(4) Read all pages
  3. (5)-(6) Make some small writes
  4. (7)-(8) Free mem and (9) exit
- After each step: pause and let memory stabilize (80 s).
- Each run is in a new VM.
- After each run the memory is allowed to stabilize.
- Each VM gets 256MB, of which 75% is filled.
- *How many concurrent VMs?*

Identical Pages

- With zero pages performance is similar.
- Reading invalidates condition C3 and C4, but not C2.
- Reads are free for sharing, otherwise performance is close.

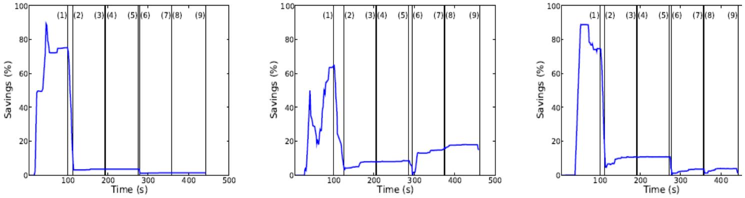


(a) Sharing (b) Patching (c) Compression

Figure 5: Workload: Identical Pages. Performance with zero pages is very similar. All mechanisms exhibit similar gains.

Random Pages

None performs well, sharing is the worst.

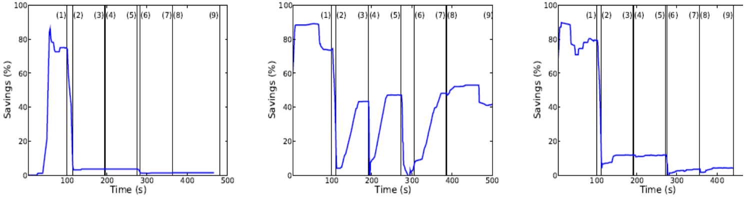


(a) Sharing (b) Patching (c) Compression

Figure 6: Workload: Random Pages. None of the mechanisms perform very well, with sharing saving the least memory.

Pages 95% Similar to an Original Page

Sharing and compression do not take advantage of similarity (compared to random pages). Patching does.



(a) Sharing (b) Patching (c) Compression

Figure 7: Workload: Similar Pages with 95% similarity. Patching does significantly better than compression and sharing.

Hypervisor Settings for Real World Workloads

To enable comparison against VMware ESX:

- Limited to one CPU (2.3 GHz Intel Xeon) due to license.
- *How much memory?*
- Same Os images.
- ESX set to most aggressive configuration (10,000 $\frac{page}{s}$), DE configured similarly. *But according to Carl Waldspurger, ESX's scan is capped at 500 $\frac{page}{s}$ per VM!*

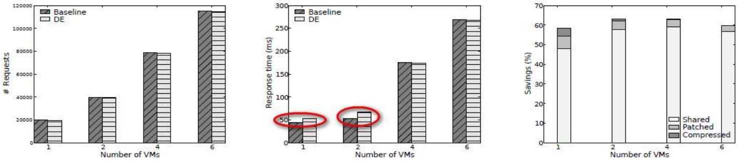
Homogeneous VMs: Xen vs. Xen+DE

Workloads: 1-6 VMs with 256MB.

- More sharing opportunities expected: PHP RUBiS on Debian. 2 client machines, each with 100 client sessions. Duration: 20 minutes.
- Less sharing opportunities expected: Linux kernel compilation.

Homogeneous VMs: Xen vs. Xen+DE

- RUBiS: Performance is unaffected, 60% of the memory is saved.
- Kernel: performance within 5%, 40% savings for 4 and more machines.
- Sharing is by design the largest memory saver.

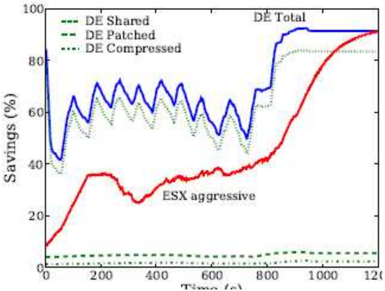


(a) Total requests handled (b) Average response time (c) Average and maximum savings

Figure 8: Difference Engine performance with homogeneous VMs running RUBiS

Homogeneous VMs: Xen+DE, ESX

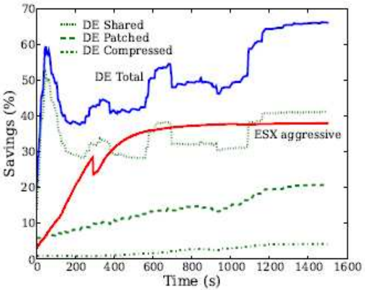
Workload: 4 VMs, each with 512MB. dbench for 10 minutes, 20 minutes stabilization.



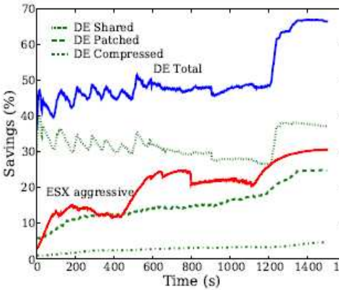
In the end ESX catches up, but during operation DE performs 1.5 times better. *ESX finds more* **sharing** *opportunities!*

Heterogeneous VMs: Xen+DE, ESX

Mixed-1: DE up to 45% better.



Mixed-2: DE X2 better.



Heterogeneous VMs: Xen+DE, ESX Performance Overhead (for Mixed-1)

- Xen+DE over Xen: up to 7%.
- ESX with aggressive (*capped!*) page sharing over ESX without page sharing: 5%.

	Kernel Compile (sec)	**Vim compile, lmbench (sec)**	**RUBiS requests**	**RUBiS response time(ms)**
Baseline	670	620	3149	1280
DE	710	702	3130	1268

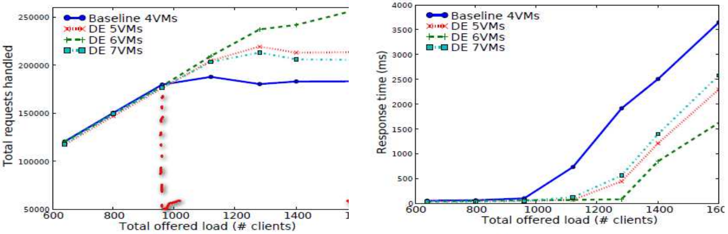
Table 3: Application performance under Difference Engine for the heterogeneous workload MIXED-1 is within 7% of the baseline.

Settings for Aggregate System Performance

- 4 cores
- 2.8GB free machine memory (excluding Dom0).
- 4 VMs and above, each allocated 650MB
- Workload: RUBiS (Java servlets implementation), 2 client machines

Aggregate Performance for Memory Over-Commit

- Xen: At 960 clients, 4 VMs use over 95% memory, some OS paging. 2 VMs with 1.2GB each do no better.
- Best DE: 6 VMs: manages 1.4 times the available memory
- Beyond 1400 clients: hypervisor paging (5000-20000 pages out, $\frac{1}{4}$ of it in)



(a) Total requests handled (b) Average response time

Conclusions and Future Work

Conclusions

- Patching and in-memory compression can bring significant savings over sharing only.
- Difference Engine outperforms (*a handicapped*) VMware ESX by 1.6-2.5 for a similar performance overhead.

Future Work:

- DE mechanisms can improve a single OS memory management.
- *Compress NRU shared pages.*
- *Protect against side channel attacks.*