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, OSDI’08

Presenter: Liran Funaro
Based on slides by Orna Agmon Ben-Yehuda and Konrad Goluchowski

Department of Computer Science
Technion — Israel Institute of Technology

Seminar on Storage Systems: Deduplication
Spring 2017
Scope

- **Backup** is not mentioned in this paper.
Scope

- **Backup** is not mentioned in this paper.
- **Rabin** and **fingerprints** are mentioned only in the related word.
  - However, **Hash** functions are used extensively.
Scope

- **Backup** is not mentioned in this paper.
- **Rabin** and **fingerprints** are mentioned only in the related word.
  - However, **Hash** functions are used extensively.
- **Deduplication** is not mentioned at all.
Scope

- **Backup** is not mentioned in this paper.
- **Rabin** and **fingerprints** are mentioned only in the related word.
  - However, **Hash** functions are used extensively.
- **Deduplication** is not mentioned at all.
- But **Delta Encoding** is mentioned.
  - Delta Encoding is essentially a **diff**.
Scope

- **Backup** is not mentioned in this paper.
- **Rabin** and **fingerprints** are mentioned only in the related word.
  - However, **Hash** functions are used extensively.
- **Deduplication** is not mentioned at all.
- But **Delta Encoding** is mentioned.
  - Delta Encoding is essentially a **diff**.
  - A diff B ⇒ Δ
Scope

- **Backup** is not mentioned in this paper.
- **Rabin** and **fingerprints** are mentioned only in the related work.
  - However, **Hash** functions are used extensively.
- **Deduplication** is not mentioned at all.
- But **Delta Encoding** is mentioned.
  - Delta Encoding is essentially a **diff**.
  - A **diff** B $\Rightarrow$ $\Delta$
  - A **patch** $\Delta$ $\Rightarrow$ B
Scope

- **Backup** is not mentioned in this paper.
- **Rabin** and **fingerprints** are mentioned only in the related word.
  - However, **Hash** functions are used extensively.
- **Deduplication** is not mentioned at all.
- But **Delta Encoding** is mentioned.
  - Delta Encoding is essentially a **diff**.
  - A **diff** B ⇒ Δ
  - A **patch** Δ ⇒ B
- Other, simpler, deduplication methods are used without formally mentioned as such.
Server Consolidation

- Instead of installing many small machines; one for each service.
- Consolidating many services on the same server.
- Server consolidation saves money and energy.
  - Reuse the same large server.
  - Low CPU utilization by individual services (e.g. 5-10%).
  - Deploy a new service within minutes.
- **Problem**: Different services can interfere with each other.
  - Using similar addresses/ports/paths.
  - Require different libraries versions, or different OSes.
- **Solution**: Put each service in its own VM.
What is Virtual machine monitor (VMM)?

Guest OS

Virtual machine monitor

Host OS

Hardware
A few samples of VMM

Xen™

vmware®

VirtualBox
Server consolidation via VMs

- Each VM runs a single service; many VMs on a single physical machine.
  - **Robustness**: single service crash will not affect the other VMs.
  - **Isolation**: a single service cannot interfere with another service data.
  - **Diversity**: each service can use its own libraries and choose different OSes.
  - **Agility**: can easily toggle services and/or migrate to a new location.
Server consolidation via VMs

- Each VM run a single service; many VMs on a single physical machine.
  - **Robustness**: single service crash will not affect the other VMs.
  - **Isolation**: a single service cannot interfere with another service data.
  - **Diversity**: each service can use its own libraries and choose different OSes.
  - **Agility**: can easily toggle services and/or migrate to a new location.

- **Infrastructure-as-a-Service (IaaS)**
  - IaaS cloud providers (Amazon, Google, etc...) rent a bundle of resources in the form of a guest virtual machine (VM) to their clients.
  - Guests can run which OS they choose or a pre-installed image by the provider.
Overcommitment

- **CPU**
  - Low CPU utilization by individual services/clients.
  - Higher CPU utilization as consolidation grows.

Memory cannot be overcommitted as easily! Paging/Swapping is slow, and introduces more challenges in virtualized environment (double paging).

Increasing the memory of the physical machine is expensive, power consuming and may not always be viable due to the machine limitation.

This makes memory the key bottleneck for VM consolidation.
Overcommitment

- **CPU**
  - Low CPU utilization by individual services/clients.
  - Higher CPU utilization as consolidation grows.

- **Memory**
  - Memory cannot be overcommitted as easily!
  - Paging/Swapping is slow, and introduces more challenges in virtualized environment (double paging).
  - Increasing the memory of the physical machine is expensive, power consuming and may not always be viable due to the machine limitation.
  - This makes memory the key bottleneck for VM consolidation.
Existing Memory Overcommitment Mechanisms

1. Paging / Swapping
   - Tragic in terms of performance.

2. Ballooning
   - Memory resource management in VMware ESX server.
   - Implemented in VMware's ESX server.
   - Well established, drivers widely available for Linux and Windows.
   - Requires memory elastic applications (not that many exists).

3. Page Sharing
   - Transparent (also Waldspurger, OSDI'02)
   - Scan through the entire memory for duplicated pages.
   - Implemented in VMware's ESX server.
   - Collaborative
   - Disco: Running commodity OSes on scalable multiprocessors.
   - (Bugnion et al., OSP'97)
   - Guest OS share its loaded code and page cache with other guests.
   - Requires guest modification (paravirtualization).
   - Benefit decrease with heterogeneous VMs.

4. Compression
   - The compression cache: Using on-line compression to extend physical memory.
   - (Douglis et al., USENIX-WTC'93)
   - Performance overhead can outweigh the memory saving due to slow hardware.
Existing Memory Overcommitment Mechanisms

1. Paging / Swapping
   - Tragic in terms of performance.

2. Ballooning
   - *Memory resource management in Vmware ESX server.* (Waldspurger, OSDI’02)
     - Implemented in VMWare’s ESX server
     - Well established, drivers widely available for Linux and Windows.
     - Requires memory elastic applications (not that many exists).
Existing Memory Overcommitment Mechanisms

1. **Paging / Swapping**
   - Tragic in terms of performance.

2. **Ballooning** *Memory resource management in VMware ESX server.* (Waldspurger, OSDI’02)
   - Implemented in VMWare’s ESX server
   - Well established, drivers widely available for Linux and Windows.
   - Requires memory elastic applications (not that many exists).

3. **Page Sharing**
   - **Transparent** (also Waldspurger, OSDI’02)
     - Scan through the entire memory for duplicated pages.
     - Implemented in VMWare’s ESX server.
   - **Collaborative** *Disco: Running commodity OSes on scalable multiprocessors.* (Bugnion et al., OSP’97)
     - Guest OS share its loaded code and page cache with other guests.
     - Requires guest modification (paravirtualization).
   - Benefit decrease with heterogeneous VMs.
Existing Memory Overcommitment Mechanisms

1. **Paging / Swapping**
   - Tragic in terms of performance.

2. **Ballooning**  
   - Memory resource management in VMware ESX server. (Waldspurger, OSDI’02)
     - Implemented in VMWare’s ESX server
     - Well established, drivers widely available for Linux and Windows.
     - Requires memory elastic applications (not that many exists).

3. **Page Sharing**
   - Transparent (also Waldspurger, OSDI’02)
     - Scan through the entire memory for duplicated pages.
     - Implemented in VMWare’s ESX server.
   - Collaborative  
     - Disco: Running commodity OSes on scalable multiprocessors. (Bugnion et al., OSP’97)
     - Guest OS share its loaded code and page cache with other guests.
     - Requires guest modification (paravirtualization).
     - Benefit decrease with heterogeneous VMs.

4. **Compression**  
   - The compression cache: Using on-line compression to extend physical memory. (Douglis et al., USENIX- WTC’93)
     - Performance overhead can outweighed the memory saving due to slow hardware.
A New Memory Overcommitment Mechanism

- Difference Engine introduces a new **sub page** sharing mechanism.
- Detecting similar, not identical, pages.
- Use **delta encoding** (patch) to represent similar pages in memory.
  - Specifically, *Xdelta*, open-source binary diff. (http://www.xdelta.org/)
- Introduces new challenges:
  - Memory is constantly changing, much more than a disk file system.
  - Time periods are not comparable with those of a backup system.
  - Memory footprint and CPU overhead should not outweigh the benefit.
Cascade of Mechanisms

In this example, two VMs have allocated five pages total.
Each initially backed by distinct pages in machine memory.

For brevity, we only show how the mapping from guest physical memory to machine memory changes; the guest virtual to guest physical mapping remains unaffected.
Cascade of Mechanisms

- **Identify**: Find identical pages.
- **Share**: Directing guest pages to the same physical page, read only.
- **Break**: Copy On Write (COW).
- **Clean**: When 0 references.
Cascade of Mechanisms

- **Identify**: Find similar pages that are not identical.
- **Patching**: Creating a patch against a reference page (Xdelta).
- **Discard**: Freeing the redundant copy.
- **Read/Write**: Rebuild the page when needed.
Cascade of Mechanisms

- Find pages that are unique and infrequently accessed.
- Compress them in memory to save space.
- Read/Write: Rebuild the page when needed.
Potential Estimate for Patching and Sharing

Ran MIXED-1
Suspended the VM after completing the benchmark.
Took memory snapshot.
Computed patches with XDelta.

1. Zero pages appear less when VMs get less memory.
2. About 14 copies of every non-unique page.
3. 50% savings. 20% saving only from sharing.
4. 77% savings. 64% for sharing+patching. 55% savings over sharing.
Mixed Real-World Workloads

- Each VM with 512 MB. Stressing memory.
- Following VMmark (VMware), VMbench (Moeller, PhD thesis)
Mixed Real-World Workloads

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

1. MIXED-1
   - **Windows**, running RUBiS (e-commerce: Apache+MySQL)
   - **Debian**, compiling Linux kernel
   - **Slackware**, compiling Vim, then running lmbench (memory, network, filesystem, signals....)
Mixed Real-World Workloads

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

1. **MIXED-1**
   - **Windows**, running RUBiS (e-commerce: Apache+MySQL)
   - **Debian**, compiling Linux kernel
   - **Slackware**, compiling Vim, then running lmbench (memory, network, filesystem, signals....)

2. **MIXED-2**
   - **Windows**, Apache with 32K static pages requested by external httperf
   - **Debian**, Sysbench (db) with 10 threads creating 100K requests
   - **Slackware**, dbench (filesystem) with 10 clients for 6 minutes, then IOZone (filesystem)
Potential Estimate for Patching and Sharing

- Ran **MIXED-1**
  - Tried many configurations.
  - This was the least savings of their attempts.
- Suspended the VM after completing the benchmark.
- Took memory snapshot.
- Computed patches with XDelta.
  - Patch limit: 2K (half a page)
  - Average patch: 1K

---

Zero pages appear less when VMs get less memory.

About 14 copies of every non-unique page.

50% savings. 20% saving only from sharing.

77% savings. 64% for sharing+patching. 55% savings over sharing.
Potential Estimate for Patching and Sharing

- Ran **MIXED-1**
- Suspended the VM after completing the benchmark.
- Took memory snapshot.
- Computed patches with XDelta. Max: 2K, Avg.:1K

<table>
<thead>
<tr>
<th>Pages</th>
<th>Initial</th>
<th>After Sharing</th>
<th>After Patching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique</td>
<td>191,646</td>
<td>191,646</td>
<td></td>
</tr>
<tr>
<td>Sharable (non-zero)</td>
<td>52,436</td>
<td>2,3577</td>
<td>1</td>
</tr>
<tr>
<td>Zero</td>
<td>149,038</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>393,120</td>
<td><strong>195,224</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Reference</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Patchable</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 Zero pages appear less when VMs get less memory.
2 About 14 copies of every non-unique page.
3 50% savings. 20% saving only from sharing.
Potential Estimate for Patching and Sharing

- Ran **MIXED-1**
- Suspended the VM after completing the benchmark.
- Took memory snapshot.
- Computed patches with XDelta. Max: 2K, Avg.:1K

<table>
<thead>
<tr>
<th>Pages</th>
<th>Initial</th>
<th>After Sharing</th>
<th>After Patching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique</td>
<td>191,646</td>
<td>191,646</td>
<td></td>
</tr>
<tr>
<td>Sharable (non-zero)</td>
<td>52,436</td>
<td>2</td>
<td>3,577</td>
</tr>
<tr>
<td>Zero</td>
<td>149,038</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>393,120</td>
<td>195,224</td>
<td>88,422</td>
</tr>
<tr>
<td>Reference</td>
<td>50,727</td>
<td></td>
<td>50,727</td>
</tr>
<tr>
<td>Patchable</td>
<td>144,497</td>
<td></td>
<td>37,695</td>
</tr>
</tbody>
</table>

1. Zero pages appear less when VMs get less memory.
2. About 14 copies of every non-unique page.
3. 50% savings. 20% saving only from sharing.
4. 77% savings. 64% for sharing+patching. 55% savings over sharing.

Presenter: Liran Funaro (Technion)
Don’t Talk the Talk If You Can’t Walk the Walk

- That was just the talk, but can walk do the walk?
- Need to identify identical and similar pages:
  - With a low memory footprint.
  - With a low computation overhead.
- Need to select potential pages for patching and compression:
  - Preferably ones that are not going to be accessed in the near future.
Identify Identical Pages (Sharing)
*On the feasibility of memory sharing.* Kloster et al., Master’s thesis

- **SuperFastHash**
- **Problem**: Hash table needs to fit in Xen’s limited heap memory (12MB)
  - Need at least one entry per page (using open addressing).
  - 4GB machine has 1,048,576 pages, yielding an 8MB hash table.
- **Solution**: Split the hash value space to 5 intervals.
  - *Constant* 5 passes, using $\frac{1}{5}$ of the hash value space each time, ignoring the rest.
  - 1.76MB hash table size (10% extra space).
  - Pass all the memory each time, only insert to table the keys on which working at that round
- Byte-by-byte comparison before actually sharing the page.
Detecting Similar Pages (Patching)

- **HashSimilarityDetector**\((k, s), c\)
- Hash \((k \cdot s)\) randomly chosen 64-byte block locations on the page.
  - Using *Rabin* fingerprint? *SHA-1*? *SuperFastHash*? Maybe modulo?
  - Was not mentioned explicitly in the paper.
- Group to \(k\) groups, each group is an index in the hash table

With \(c = 1\), we only store the first page found with each signature.
With \(c > 1\), we keep multiple pages in the hash table for each index.

When trying to build a patch, Difference Engine computes a patch between all matching pages and chooses the best one.

Smaller \(k, s, c\) ⇒ less resources used.

So what parameters do we choose?
Examples of $\text{HashSimilarityDetector}(k, s)$

- Group to $k$ groups, each group is an index in the hash table

<table>
<thead>
<tr>
<th></th>
<th>1, 1</th>
<th>1, 2</th>
<th>2, 1</th>
<th>2, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
Detecting Similar Pages (Patching)

- **HashSimilarityDetector**\((k, s), c\)
- Hash \((k \cdot s)\) randomly chosen 64-byte block locations on the page.
  - Using *Rabin* fingerprint? *SHA-1*? *SuperFastHash*? Maybe modulo?
  - Was not mentioned explicitly in the paper.
- Group to \(k\) groups, each group is an index in the hash table
- Pages that match at least one of the \(k\) blocks are chosen as candidates.
Detecting Similar Pages (Patching)

- **HashSimilarityDetector**\((k, s), c\)
- Hash \((k \cdot s)\) randomly chosen 64-byte block locations on the page.
  - Using *Rabin* fingerprint? *SHA-1*? *SuperFastHash*? Maybe modulo?
  - Was not mentioned explicitly in the paper.
- Group to \(k\) groups, each group is an index in the hash table
- Pages that match at least one of the \(k\) blocks are chosen as candidates.
- \(c\): number of different pages stored for each key
  - With \(c = 1\), we only store the first page found with each signature.
  - With \(c > 1\), we keep multiple pages in the hash table for each index.
  - When trying to build a patch, Difference Engine computes a patch between all matching pages and chooses the best one.
- Smaller \(k, s, c\) ⇒ less resources used.
- So what parameters do we choose?
Savings from Patching Only\(^1\) as Function of \(k, s, c\)

\[\text{mem savings} = 100 \times \left(1 - \frac{\text{Total memory actually used}}{\text{Total memory allocated to VMs}}\right)\]

\(^1\)Savings from patching after all identical pages have been shared.
Chosen: \textbf{HashSimilarityDetector}(2, 1), 1

- 2 hash keys of single locations, 1 stored page.
- Similarity Hash Table is also statically allocated in Xen heap (12MB).
- Each hash table entry is the size of a pointer (4 bytes).
  - 32-bit hashes yields \(\sim 25\%\) saving for MIXED-1, but 16GB hash table.
  - 18-bit hashes yields \(\sim 20\%\) saving for MIXED-1, but 1MB hash table.
- \(\Rightarrow\) We use 18-bit hashes.
Compression

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

- Safety net
- Involves disk I/O - slow
- Swapped out pages cannot be shared or referenced for patching
We Need a Policy

- We need a policy to decide when a page is eligible for:
  - Sharing
  - Use as Patch-Reference
  - Patching
  - Compression
  - Paging
We Need a Policy

- We need a policy to decide when a page is eligible for:
  - Sharing
  - Use as Patch-Reference
  - Patching
  - Compression
  - Paging

- Default policy: Not-Recently-Used\(^1\) (NRU)
  - (C1) Recently modified
  - (C2) Recently read only
  - (C3) Recently nothing
  - (C4) Nothing for several scans

\(^1\)Recently = since last scan.
We Need a Policy

- We need a policy to decide when a page is eligible for:
  - Sharing
  - Use as Patch-Reference
  - Patching
  - Compression
  - Paging

- Default policy: Not-Recently-Used¹ (NRU)
  - (C1) Recently modified $\Rightarrow$ Ignore
  - (C2) Recently read only
  - (C3) Recently nothing
  - (C4) Nothing for several scans

¹Recently = since last scan.
We Need a Policy

- We need a policy to decide when a page is eligible for:
  - Sharing
  - Use as Patch-Reference
  - Patching
  - Compression
  - Paging

- Default policy: Not-Recently-Used\(^1\) (NRU)
  - (C1) Recently modified \(\Rightarrow\) Ignore
  - (C2) Recently read only \(\Rightarrow\) Share/Patch-Reference
  - (C3) Recently nothing
  - (C4) Nothing for several scans

\(^1\)Recently = since last scan.
We Need a Policy

- We need a policy to decide when a page is eligible for:
  - Sharing
  - Use as Patch-Reference
  - Patching
  - Compression
  - Paging

- Default policy: Not-Recently-Used\(^1\) (NRU)
  - (C1) Recently modified \(\Rightarrow\) Ignore
  - (C2) Recently read only \(\Rightarrow\) Share/Patch-Reference
  - (C3) Recently nothing \(\Rightarrow\) Share/Patch
  - (C4) Nothing for several scans

\(^1\)Recently = since last scan.
We Need a Policy

- We need a policy to decide when a page is eligible for:
  - Sharing
  - Use as Patch-Reference
  - Patching
  - Compression
  - Paging

- Default policy: Not-Recently-Used\(^1\) (NRU)
  - (C1) Recently modified ⇒ Ignore
  - (C2) Recently read only ⇒ Share/Patch-Reference
  - (C3) Recently nothing ⇒ Share/Patch
  - (C4) Nothing for several scans ⇒ Anything

\(^1\)Recently = since last scan.
We Need a Policy

- We need a policy to decide when a page is eligible for:
  - Sharing
  - Use as Patch-Reference
  - Patching
  - Compression
  - Paging

- Default policy: Not-Recently-Used\(^1\) (NRU)
  - (C1) Recently modified \(\Rightarrow\) Ignore
  - (C2) Recently read only \(\Rightarrow\) Share/Patch-Reference
  - (C3) Recently nothing \(\Rightarrow\) Share/Patch
  - (C4) Nothing for several scans \(\Rightarrow\) Anything

- Alternative policies:
  - Consider all pages for anything \(\Rightarrow\) insignificant excess saving.
  - Compression before patching \(\Rightarrow\) slightly less savings, less performance overhead.

\(^1\)Recently = since last scan.
Clock

- Each invocation is at least 4 seconds apart.

Clock Invocation Detailed:

- Read and resets Read, Modified bits
  - Yielding a limited-size list of NRU pages.
- (×5) Scans pages looking for identical and similar pages.
  - Scan a small portion of each VM’s memory in turn for fairness.
- Clear sharing hash table before every scan.
- Similarity Hash Table is cleared only before the first scan.
  - Allows finding similarity between keys from different passes
  - Only patch pages that their hash falls in the current interval to avoid patch a page that might be eligible for sharing.
  - Easier to discard previous data, than trap page writes.

- Resort to compression
  - Compression postponed till after all pages are checked for similarity (prevents sharing patching)
  - All the remaining pages that are marked C4 are compressed.
Default Evaluation Setup

- 4 cores (dual processor, dual core 2.33 GHz Intel Xeon)
- Page size 4K
Clock Performance - Lifetime of Patched/Compressed Pages

- A good clock should give high lifetimes to compressed and patched pages, which are costly to access.
- Performance of heterogeneous workload close to homogeneous workload.
- *Good performance?*
Times of Individual Operations

Using micro benchmarks.

<table>
<thead>
<tr>
<th>Function</th>
<th>Mean execution time ($\mu s$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>share_pages</td>
<td>6.1</td>
</tr>
<tr>
<td>cow_break</td>
<td>25.1</td>
</tr>
<tr>
<td>compress_page</td>
<td>29.7</td>
</tr>
<tr>
<td>decompress</td>
<td>10.4</td>
</tr>
<tr>
<td>patch_page</td>
<td>338.1</td>
</tr>
<tr>
<td>unpatch</td>
<td>18.6</td>
</tr>
<tr>
<td>swap_out_page</td>
<td>48.9</td>
</tr>
<tr>
<td>swap_in_page</td>
<td>7151.6</td>
</tr>
</tbody>
</table>

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

Presenter: Liran Funaro (Technion)
4 steps:

1. (1)-(2) Allocate pages [zero / random / identical / similar\textsuperscript{1}].
2. (3)-(4) Read all pages.
3. (5)-(6) Make some small writes to all pages.
4. (7)-(8) Free memory 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.

\textsuperscript{1}Similar but not identical.
Identical Pages

- With zero pages performance is similar.
- Reading invalidates patches and compression, but not sharing.
- Reads are free for sharing, otherwise performance is close.

(1)-(2) allocate, (3)-(4) read, (5)-(6) write, (7)-(8) free, (9) exit.
Random Pages

- None performs well.
- Sharing is the worst.

(1)-(2) allocate, (3)-(4) read, (5)-(6) write, (7)-(8) free, (9) exit.
- Sharing and compression do not take advantage of similarity.
- Patching does (compared to random pages).

\(1\)-(2) allocate, (3)-(4) read, (5)-(6) write, (7)-(8) free, (9) exit.
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? Is there overcommitment?*
- Use matching OS images for Xen and ESX.
- ESX set to most aggressive configuration (10,000 \(\text{page/s}\))\(^1\), DE configured similarly.

\(^1\)According to Carl Waldspurger, ESX’s scan is capped at 500 \(\text{page/s}\) 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
Homogeneous VMs: Xen+DE vs. 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 $\times 1.5$ better.

ESX finds more sharing opportunities!
Heterogeneous VMs: Xen+DE vs. ESX

(a) MIXED-1: DE up to 45% better.

(b) MIXED-2: DE $\times 2$ better.

- In steady state, DE delivers $\times 1.6 - 2.5$ more savings than ESX.
Heterogeneous VMs: Xen+DE vs. ESX
Performance Overhead (for MIXED-1)

- Xen+DE over Xen: within 7% of the baseline.
- ESX with aggressive (capped!) page sharing over ESX without page sharing: 5%.

<table>
<thead>
<tr>
<th></th>
<th>Kernel Compile (sec)</th>
<th>Vim compile, Imbench (sec)</th>
<th>RUBiS requests</th>
<th>BUBiS response time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>670</td>
<td>620</td>
<td>3149</td>
<td>1280</td>
</tr>
<tr>
<td>DE</td>
<td>710</td>
<td>702</td>
<td>3130</td>
<td>1268</td>
</tr>
</tbody>
</table>

Presenter: Liran Funaro (Technion)
Settings for Aggregate System Performance
Finally overcommitment!

- 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 Overcommitment

- **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 $\times 1.4$ the available memory
- Beyond 1400 clients: hypervisor paging (5,000-20,000 pages out, $\frac{1}{4}$ of it in)
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 \( \times 1.6 - 2.5 \) for a similar performance overhead.

- **Future Work:**
  - DE mechanisms can improve a single OS memory management.
Questions?
Changes to Xen

- 14.5K lines added + 20K lines for porting existing libraries.
- Changes mainly in guest physical to machine table, and in the shadow page tables
- Difference Engine (DE) not in effect during the VM’s boot (real-mode), 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.
  - Every 10 seconds, unmap unused pages.
- Block allocator - to efficiently manage storage of compressed and shared pages (consume less than one page).
P2M Mapping

- Copy those R/M bits in the guest-physical to host-physical (P2M) map, from the shadow page tables.
- Add 2 additional bits to the P2M to support (C4) (count consecutive (C3)).
- Races when updating P2M and shadow table:
  - Locking (pausing the domain) only when building patch and replacing page.
  - Other races only result in larger patches (using inadequate page for patching).
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).