PageForge: A Near-Memory Content-Aware Page-Merging Architecture

Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas

University of Illinois at Urbana-Champaign
Motivation: Server Consolidation in the Cloud
Motivation: Server Consolidation in the Cloud

How to consolidate Main Memory?
Content-Aware Page Merging or Page Deduplication
Content-Aware Page Merging or Page Deduplication

• Hypervisors search the address space and merge identical pages
Content-Aware Page Merging or Page Deduplication

- Hypervisors search the address space and merge identical pages
Content-Aware Page Merging or Page Deduplication

- Hypervisors search the address space and merge identical pages
Problem: **Overhead of Software Page Merging**

- Search hundreds of millions of pages
- Latency sensitive applications get disrupted
- RedHat’s SW page merging (KSM) has execution overhead:
  - Average mean latency overhead: 68%
  - Average tail latency overhead: 136%
Contribution: **PageForge**

- First solution for hardware-assisted content-aware page merging
- General, effective, minimal hypervisor involvement & hardware mods
- Reduced overhead vs state-of-the-art software:
  - Mean latency 68% → 10%
  - Tail latency 136% → 11%
- Same memory savings as software: 48% → Twice #VMs
- Novel use of ECC for page content characterization
Content Duplication in the Cloud
Background: Cloud

Infrastructure
Background: Cloud
Background: Cloud
Background: Cloud

- Infrastructure
  - Host Operating System
    - Hypervisor
      - Guest OS
Background: Cloud

- Infrastructure
  - Hypervisor
  - Host Operating System
  - Guest OS
  - Libs
Background: Cloud
Background: Cloud
Background: Content Duplication
Background: Content Duplication

Diagram:
- Hypervisor
- Host Operating System
- Infrastructure
- Data
- python
- ubuntu®
- python
- ubuntu®
- python
- ubuntu®
Background: Content Duplication
Background: Content Duplication
Background: Content Duplication
Background: Content Duplication

Page Merging

Content Duplication

Data
Libs
Guest OS

Data
Libs
Guest OS

Data
Libs
Guest OS

Hypervisor

Host Operating System

Infrastructure
Our Proposal: Content Deduplication in HW

Page Merging

Content Duplication
Software-based Content-Aware Page Merging
Software-Based Content-Aware Page Merging

Pool of Pages to Scan

Pool of Stable Pages
Software-Based Content-Aware Page Merging

Pool of Pages to Scan

Pool of Stable Pages

Candidate Page

Compare Page
Software-Based Content-Aware Page Merging

Main Memory

Candidate Page

4KB

Compare Page

4KB

Memory Ctrl

L3 $

L2 $

L1 $

Core
Software-Based Content-Aware Page Merging

Main Memory

Candidate Page

4KB 64B

Compare Page

4KB 64B

Memory Ctrl

L3 $

L2 $ L1 $ Core
Software-Based Content-Aware Page Merging

Pool of Pages to Scan

4KB

Candidate Page

4KB

4KB

4KB

Compare Page

Pool of Stable Pages

4KB

4KB

4KB

4KB
Software-Based Content-Aware Page Merging

Pool of Pages to Scan

4KB
Candidate Page

4KB

4KB

4KB

Compare Page

Pool of Stable Pages

4KB

4KB

4KB

4KB
Why Page-Merging is expensive?

- Search hundreds of millions of pages
  - Many core cycles consumed
  - Caches get polluted
  - Latency sensitive applications get disrupted
Software-Based Content-Aware Page Merging

- Optimization: Identify if the candidate page has recently changed
- If a page changes too often → Not a good candidate

![Diagram]

Candidate Page

- 1KB
- 4KB

Hash Key

JHash
PageForge: Hardware Assisted Page-Merging
PageForge Main Idea

- Hardware engine in the memory controller
  - Compares pages
  - Generates hashes

- Advantages:
  - No core utilization
  - No cache pollution
PageForge Main Idea

- Hardware engine in the memory controller
- Compares pages
- Generates hashes

Advantages:
- No core utilization
- No cache pollution
PageForge @ The Memory Controller

- Read Req Buffer
- Write Req Buffer
- Scheduler (FR-FCFS)
- Command Generation
- Memory Interface
- Memory Controller

On-Chip Network
- Control
- Comparator
- Scan Table
- PageForge
- Read Data Buffer
- ECC Dec
- ECC Enc
- Write Data Buffer
PageForge Operation
PageForge Operation

- Loads in Scan Table:
  - Candidate page PPN
  - PPNs of pages to compare to candidate
PageForge Operation

**Operating System**
- Loads in Scan Table:
  - Candidate page PPN
  - PPNs of pages to compare to candidate

**PageForge Hardware**
- Autonomous performs:
  - Sequence of comparisons
  - Generation of the hash key for candidate page
PageForge Operation

In Hardware
PageForge Operation

Search Pool of Stable Pages

Pick Candidate Page

In Hardware
PageForge Operation

- Search Pool of Stable Pages
  - Pick Candidate Page
  - Match Found?
    - Yes
      - OS Merges Pages

In Hardware
PageForge Operation

Search Pool of Stable Pages → Match Found?

- Yes → Pick Candidate Page
- No → Old Key = New Key?

- No → OS Merges Pages
- Yes → In Hardware
PageForge Operation

Search Pool of Stable Pages

Match Found?

Yes

OS Merges Pages

No

Old Key = New Key?

Yes

Search Pool of Unprotected Pages

No

Match Found?

Yes

In Hardware
PageForge Operation

1. Search Pool of Stable Pages
2. Pick Candidate Page
3. OS Merges Pages
4. Match Found?
   - Yes: In Hardware
   - No: Old Key = New Key?
     - Yes: Search Pool of Unprotected Pages
     - No: Insert Candidate in Unprotected Pool
5. Match Found?
   - Yes: Insert Candidate in Unprotected Pool
   - No: Search Pool of Stable Pages
PageForge Operation

1. Search Pool of Stable Pages
2. Match Found?
   - No
   - Yes
     - Old Key = New Key?
     - No
     - Yes
     - Search Pool of Unprotected Pages
   - Yes
     - Match Found?
     - No
     - Yes
       - Insert Candidate in Unprotected Pool
       - No
3. OS Merges Pages
4. Pick Candidate Page
5. In Hardware
Eliminating The Cost of Hash Keys
Hash Key for Page Content Characterization
Hash Key for Page Content Characterization

• If page changed recently, key *may* be different
Hash Key for Page Content Characterization

- If page changed recently, key *may* be different

- KSM: JHash $\rightarrow$ Serial computation + 1KB of data
Hash Key for Page Content Characterization

• If page changed recently, key *may* be different

• KSM: JHash → Serial computation + 1KB of data

• PageForge: ECC → Parallel computation + 256B of data
PageForge @ The Memory Controller

![Diagram of PageForge Memory Controller]

- **Read Req Buffer**
- **Write Req Buffer**
- **Scheduler (FR-FCFS)**
- **Command Generation**
- **Memory Interface**
- **On-Chip Network**
- **Off-Chip Memory Interface**
- **Control**
- **Comparator**
- **Scan Table**
- **Read Data Buffer**
- **ECC Enc**
- **ECC Dec**
- **Write Data Buffer**
- **Memory Controller**
Proposal: ECC for Page Content Characterization
Proposal: ECC for Page Content Characterization

- Break a 4KB page into 4 segments
Proposal: ECC for Page Content Characterization

- Break a 4KB page into 4 segments
- Pick a random cache line within each segment
Proposal: ECC for Page Content Characterization

- Break a 4KB page into 4 segments
- Pick a random cache line within each segment
- Select the 8-bit ECC of the least significant QWORD
Proposal: ECC for Page Content Characterization

- Break a 4KB page into 4 segments
- Pick a random cache line within each segment
- Select the 8-bit ECC of the least significant QWORD
Proposal: ECC for Page Content Characterization

- Break a 4KB page into 4 segments
- Pick a random cache line within each segment
- Select the 8-bit ECC of the least significant QWORD

Software JHash = 1KB
Proposal: ECC for Page Content Characterization

- Break a 4KB page into 4 segments
- Pick a random cache line within each segment
- Select the 8-bit ECC of the least significant QWORD
Proposal: ECC for Page Content Characterization

- Break a 4KB page into 4 segments
- Pick a random cache line within each segment
- Select the 8-bit ECC of the least significant QWORD

ECC Hash = 256 Bytes
Proposal: ECC for Page Content Characterization

- Break a 4KB page into 4 segments
- Pick a random cache line within each segment
- Select the 8-bit ECC of the least significant QWORD

75% Memory Footprint Reduction for Hash Key Generation!
Evaluation
Simulation Setup

- 2GHz, 10-core, 16GB DRAM
- One VM per core
- Ubuntu 16.04 Host, Ubuntu Cloud Guest
- InHouse simulator: Simics + SST + DRAMSim2
- TailBench Suite benchmarks
Configurations

- **Baseline**: Page-merging is disabled
- **KSM**: RedHat’s implementations shipped with the current Linux
- **PageForge**: Hardware-assisted page-merging
Memory Savings – w/o vs w/ Page Merging

- **Img-Dnn**
- **Masstree**
- **Moses**
- **Silo**
- **Sphinx**
- **Average**

<table>
<thead>
<tr>
<th></th>
<th>w/o Page Merging</th>
<th>w/ Page Merging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Img-Dnn</td>
<td>100</td>
<td>52</td>
</tr>
<tr>
<td>Masstree</td>
<td>100</td>
<td>52</td>
</tr>
<tr>
<td>Moses</td>
<td>100</td>
<td>52</td>
</tr>
<tr>
<td>Silo</td>
<td>100</td>
<td>52</td>
</tr>
<tr>
<td>Sphinx</td>
<td>100</td>
<td>52</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>100</strong></td>
<td><strong>48%</strong></td>
</tr>
</tbody>
</table>
Memory Savings – w/o vs w/ Page Merging

PageForge Achieves Memory Savings of 48%
Memory Savings – w/o vs w/ Page Merging

PageForge Achieves Memory Savings of 48% Twice #VMs!!
Memory Savings – w/o vs w/ Page Merging

![Graph showing memory savings comparison with and without page merging for different applications: Unmergeable, Mergeable Zero, and Mergeable Non-Zero. The x-axis represents different applications (Img-Dnn, Masstree, Moses, Silo, Sphinx, Average), and the y-axis represents the percentage of pages. The graph compares 'W/O Page-Merging' and 'W/ Page-Merging' scenarios.]
Memory Savings – w/o vs w/ Page Merging

- Unmergeable
- Mergeable Zero
- Mergeable Non-Zero

Pages %

W/O Page-Merging

W/ Page-Merging

Img-Dnn

Masstree

Moses

Silo

Sphinx

Average

Average
Memory Savings – w/o vs w/ Page Merging

Unmergeable | Mergeable Zero | Mergeable Non-Zero

Pages %

48%
Mean Latency of Requests

<table>
<thead>
<tr>
<th></th>
<th>Img-Dnn</th>
<th>Masstree</th>
<th>Moses</th>
<th>Silo</th>
<th>Sphinx</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized Mean Latency</td>
<td>0.5</td>
<td>1.0</td>
<td>1.5</td>
<td>2.0</td>
<td>2.5</td>
<td>2.0</td>
</tr>
</tbody>
</table>

KSM
PageForge
72
Mean Latency of Requests

- Baseline

<table>
<thead>
<tr>
<th>System</th>
<th>Normalized Mean Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Img-Dnn</td>
<td>1</td>
</tr>
<tr>
<td>Masstree</td>
<td>1</td>
</tr>
<tr>
<td>Moses</td>
<td>1</td>
</tr>
<tr>
<td>Silo</td>
<td>1</td>
</tr>
<tr>
<td>Sphinx</td>
<td>1</td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
</tr>
</tbody>
</table>

KSM, PageForge
Mean Latency of Requests

Baseline vs KSM

Normalized Mean Latency

- Img-Dnn
- Masstree
- Moses
- Silo
- Sphinx
- Average

PageForge 74
Mean Latency of Requests

- Baseline
- KSM
- PageForge

Average Normalized Mean Latency

- KSM: 68%
- PageForge: 10%
Mean Latency of Requests

PageForge Reduces Mean Latency Overhead From 68% Down to 10%!
Tail Latency of Requests

- **Baseline**, **KSM**, and **PageForge** categories are represented.

### Latency per Component

- **Img-Dnn**
  - Baseline: 0.5
  - KSM: 0.5
  - PageForge: 1.0

- **Masstree**
  - Baseline: 2.0
  - KSM: 2.0
  - PageForge: 5.0

- **Moses**
  - Baseline: 1.5
  - KSM: 1.5
  - PageForge: 1.5

- **Silo**
  - Baseline: 1.0
  - KSM: 5.18
  - PageForge: 5.18

- **Sphinx**
  - Baseline: 0.5
  - KSM: 0.5
  - PageForge: 0.5

- **Average**
  - Baseline: 1.0
  - KSM: 2.1
  - PageForge: 2.1

**Associations and Values**

- **Baseline**: 5.18% increase
- **KSM**: 136% increase
- **PageForge**: 11% increase
PageForge Reduces Tail Latency Overhead From 136% Down to 11%!
Also in the Paper

- Software interface to PageForge
- Interaction with the cache coherence protocol
- Alternative designs
- Bandwidth analysis
- ECC vs Jhash → ECC keys add negligible collisions
- Power and area at 22nm is negligible
Takeaway: **PageForge**

- First solution for hardware-assisted content-aware page merging
- General, effective, minimal hypervisor involvement & hardware mods
- Reduced overhead vs state-of-the-art software:
  - Mean latency 68% → 10%
  - Tail latency 136% → 11%
- Same memory savings as software: 48% → Twice #VMs
- Novel use of ECC for content characterization → 75% less memory footprint