Large Pages May Be Harmful on NUMA Systems

Fabien Gaud  
Simon Fraser University

Baptiste Lepers  
CNRS

Jeremie Decouchant  
Grenoble University

Justin Funston  
Simon Fraser University

Alexandra Fedorova  
Simon Fraser University

Vivien Quéma  
Grenoble INP
Virtual-to-physical translation is done by the TLB and page table.

Typical TLB size: 1024 entries (AMD Bulldozer), 512 entries (Intel i7).
Virtual-to-physical translation is done by the TLB and page table.

Typical TLB size: 1024 entries (AMD Bulldozer), 512 entries (Intel i7).
Large pages known advantages & downsides

Known advantages:
- Fewer TLB misses
- Fewer page allocations (reduces contention in the kernel memory manager)

<table>
<thead>
<tr>
<th>Page size</th>
<th>512 entries coverage</th>
<th>1024 entries coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB (default)</td>
<td>2MB</td>
<td>4MB</td>
</tr>
<tr>
<td>2MB</td>
<td>1GB</td>
<td>2GB</td>
</tr>
<tr>
<td>1GB</td>
<td>512GB</td>
<td>1024GB</td>
</tr>
</tbody>
</table>

Known downsides:
- Increased memory footprint
- Memory fragmentation
New observation: large pages may hurt performance on NUMA machines

Performance improvement of THP (2M pages) over 4K pages

24-core machine

64-core machine
Remote memory accesses hurt performance
Machines are NUMA

Contention hurts performance even more.

Node 1

Node 2

Node 3

1200 cycles!
Large pages on NUMA machines (1/2)

```c
void *a = malloc(2MB);
```

With 4K pages, load is balanced.
Large pages on NUMA machines (1/2)

With 2M pages, data are allocated on 1 node => contention.

Node 0

Node 1

Node 2

Node 3

void *a = malloc(2MB);
With 2M pages, data are allocated on 1 node => contention.
Performance example (1/2)

<table>
<thead>
<tr>
<th>App.</th>
<th>Perf. increase THP/4K (%)</th>
<th>% of time spent in TLB miss 4K (%)</th>
<th>% of time spent in TLB miss 2M (%)</th>
<th>Imbalance 4K (%)</th>
<th>Imbalance 2M (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.D</td>
<td>-43</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>59</td>
</tr>
<tr>
<td>SSCA.20</td>
<td>17</td>
<td>15</td>
<td>2</td>
<td>8</td>
<td>52</td>
</tr>
<tr>
<td>SpecJBB</td>
<td>-6</td>
<td>7</td>
<td>0</td>
<td>16</td>
<td>39</td>
</tr>
</tbody>
</table>

Using large pages, 1 node is overloaded in CG, SSCA and SpecJBB. Only SSCA benefits from the reduction of TLB misses.
Large pages on NUMA machines (2/2)

void *a = malloc(1.5MB); // node 0
void *b = malloc(1.5MB); // node 1

Page-level false sharing reduces the maximum achievable locality.
Performance example (2/2)

<table>
<thead>
<tr>
<th>App.</th>
<th>Perf. increase THP/4K (%)</th>
<th>Local Access Ratio 4K (%)</th>
<th>Local Access Ratio 2M (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UA.C</td>
<td>-15</td>
<td>88</td>
<td>66</td>
</tr>
</tbody>
</table>

The locality decreases when using large pages.
Can existing memory management algorithms solve the problem?
Existing memory management algorithms do not solve the problem

We run the application with Carrefour[1], the state-of-the-art memory management algorithm. Carrefour monitors memory accesses and places pages to minimize imbalance and maximize locality.

Carrefour solves imbalance / locality issues on some applications

But does not improve performance on some other applications (hot pages or page-level false sharing)

We need a better memory management algorithm
Our solution – Carrefour-LP

- Built on top of Carrefour.
- By default, 2M pages are activated.
- Two components that run every second:

<table>
<thead>
<tr>
<th>Reactive component</th>
<th>Conservative component</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Splits 2M pages</strong></td>
<td><strong>Promotes 4K pages</strong></td>
</tr>
</tbody>
</table>
| Detects and removes “hot pages” and page-level “false sharing”.
| **Deactivate 2M page allocation**   | When the time spent handling TLB misses is high.      |
|                                     | **Forces 2M page allocation**                         |
|                                     | In case of contention in the page fault handler.      |

- We show in the paper that the two components are required.
Implementation

Reactive component (splits 2M pages)

Sample memory accesses using IBS

A page represents more than 5% of all accesses and is accessed from multiple nodes?

YES

Split and interleave the hot page
Implementation

Reactive component (splits 2M pages)

Sample memory accesses using IBS

- Compute observed local access ratio (LAR1)
- Compute the LAR that would have been obtained if each page was placed on the node that accessed it the most.

LAR1 can be significantly improved?

YES

Run carrefour

NO

- Compute the LAR that would have been obtained if each page was split and then placed on the node that accessed it the most.

LAR1 can be significantly improved?

YES

Split all 2M pages and run carrefour

NO
Implementation challenges

Reactive component (splits 2M pages)

Sample memory accesses using IBS

- Compute observed local access ratio (LAR1)
- Compute the LAR that would have been obtained if each page was placed on the node that accessed it the most (without splitting).

LAR1 can be significantly improved?

- YES: Run carrefour

LAR1 can be significantly improved?

- YES: Split all 2M pages and run carrefour

IMPRESSIVE

NO

COSTLY

COSTLY

IMPRECISE
Implementation challenges

Reactive component (splits 2M pages)

• We only have few IBS samples.

• The LAR with “2M pages split into 4K pages” can be wrong.

• We try to be conservative by running Carrefour first and only splitting pages when necessary (splitting pages is expensive).

• Predicting that splitting a 2M page will increase TLB miss rate is too costly. This is why the conservative component is required.
Implementation

Conservative component

- Monitor time spent in TLB misses (hardware counters)
  - > 5%
    - YES: Cluster 4K pages and force 2M pages allocation

- Monitor time spent in page fault handler (kernel statistics)
  - > 5%
    - YES: Force 2M page allocation
Evaluation

- Carrefour-2M over Linux 4K
- Reactive over Linux 4K
- Conservative over Linux 4K
- Carrefour-LP over Linux 4K

**24-core machine**

**64-core machine**

Perf. Improvement relative to 4K (%)
Conclusion

• Large pages can hurt performance on NUMA systems.

• We identified two new issues when using large pages on NUMA systems: “hot pages” and “page-level false sharing”.

• We designed a new algorithm, Carrefour-LP. On the set of applications:
  • 46% better than Linux
  • 50% better than THP.

(The full set of applications is available in the paper.)

• Overhead:
  • Less than 3% CPU overhead.

• Carrefour-LP restores the performance when it was lost due to large pages and makes their benefits accessible to applications.
Questions?
## Performance example

<table>
<thead>
<tr>
<th>App.</th>
<th>Perf. increase THP/4K</th>
<th>Time spent in page fault handler 4K</th>
<th>Time spent in page fault handler 2M</th>
<th>Local access ratio 4K (%)</th>
<th>Local Access ratio 2M (%)</th>
<th>Imbalance 4K (%)</th>
<th>Imbalance 2M (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.D</td>
<td>-43</td>
<td>2200ms (0.1%)</td>
<td>450ms (0.1%)</td>
<td>40</td>
<td>36</td>
<td>1</td>
<td>59</td>
</tr>
<tr>
<td>UA.C</td>
<td>-15</td>
<td>100ms (0.2%)</td>
<td>50ms (0.1%)</td>
<td>88</td>
<td>66</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>WR</td>
<td>109</td>
<td>8700ms (38%)</td>
<td>3700ms (32%)</td>
<td>50</td>
<td>55</td>
<td>147</td>
<td>136</td>
</tr>
<tr>
<td>SSCA. 20</td>
<td>17</td>
<td>90ms (0%)</td>
<td>150ms (0%)</td>
<td>25</td>
<td>26</td>
<td>8</td>
<td>52</td>
</tr>
<tr>
<td>SpecJB B</td>
<td>-6</td>
<td>8400ms (2%)</td>
<td>5900ms (1.5%)</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td>39</td>
</tr>
</tbody>
</table>