HawkEye: Efficient Fine-grained OS Support for Huge Pages

Ashish Panwar¹, Sorav Bansal², K. Gopinath¹

Indian Institute of Science (IISc), Bangalore¹

Indian Institute of Technology, Delhi ²
Virtual address space
Virtual address space

Physical address space
Virtual address space

Physical address space
Virtual address space

Physical address space
Too much TLB pressure!

Virtual address space

Physical address space
Virtual address space

Physical address space

TLB
Physical address space

Virtual address space

Huge pages

Fewer misses

TLB
OS Challenges

❑ Complex trade-offs
  • Memory bloat vs. performance
  • Page fault latency vs. the number of page faults

❑ Challenges due to (external) fragmentation
  • How to leverage limited memory contiguity
  • Fairness in huge page allocation
Memory bloat vs. performance
Internal fragmentation

aggressive allocation

![Diagram of virtual memory and physical memory with huge page mapping]
**Internal fragmentation**

**aggressive allocation**

- Virtual memory
- Physical memory
- huge page mapping

**conservative allocation**

- Virtual memory
- Physical memory
- base page mappings
Internal fragmentation

aggressive allocation

unused pages

conservative allocation

huge page mapping

base page mappings

Virtual memory

Physical memory

unused pages

Virtual memory

Physical memory

Internal fragmentation

aggressive allocation

unused pages

conservative allocation

huge page mapping

base page mappings

Virtual memory

Physical memory
Internal fragmentation

aggressive allocation

unused pages

huge page mapping

bloat

conservative allocation

base page mappings
Internal fragmentation

aggressive allocation

unused pages

Conservative allocation

bloat

Lower TLB reach (impacts performance)
Bloat vs. performance

- **Aggressive**
  - Higher perf
  - Higher bloat

- **Conservative**
  - Lower perf
  - Lower bloat
Latency vs. # page faults
Find a page

4-KB
- Find a page, zero-fill

4-KB

pre ➔ zero-fill
- Find a page, zero-fill, map

4-KB

pre  →  zero-fill  →  post
Find a page, zero-fill, map

4-KB

pre ➔ zero-fill ➔ post

25%
- Find a page, zero-fill, map

4-KB: pre → zero-fill → post

2-MB: pre

25%
- Find a page, zero-fill, map

4-KB:
- pre
- zero-fill
- post

2-MB:
- pre
- zero-fill
- Find a page, zero-fill, map

4-KB

25%

2-MB

pre → zero-fill → post

pre → zero-fill → post

25%
- Find a page, zero-fill, map

4-KB
- pre -> zero-fill -> post

2-MB
- pre -> zero-fill -> post

25%

dominated by zero-filling (97%)
Latency vs. # page faults

- **Aggressive**
  - High latency
  - Fewer faults

- **Conservative**
  - Low latency
  - Higher faults
Current systems favor opposite ends of the design spectrum

- FreeBSD is conservative (compromise on performance)
- Linux is throughput-oriented (compromise on latency and bloat)

<table>
<thead>
<tr>
<th></th>
<th>FreeBSD</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory bloat</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Performance</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Allocation latency</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td># page faults</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>
Ingens (OSDI’16)

- **Asynchronous** allocation
  - Huge pages allocated in the background

- **Utilization-threshold** based allocation
  - Tunable bloat vs. performance
  - Adaptive based on memory pressure

- Fairness driven by **per-process fairness metric**
  - Heuristic based on past behavior
Ingens (OSDI’16)

- **Asynchronous allocation**
  - Huge pages allocated in the background

- **Utilization-threshold** based allocation
  - Tunable bloat vs. performance
  - Adaptive based on memory pressure

- Fairness driven by **per-process fairness metric**
  - Heuristic based on past behavior

low latency
too many page faults
Ingens (OSDI’16)

- **Asynchronous** allocation
  - Huge pages allocated in the background

- **Utilization-threshold** based allocation
  - Tunable bloat vs. performance
  - Adaptive based on memory pressure

- Fairness driven by **per-process fairness metric**
  - Heuristic based on past behavior

---

low latency

too many page faults

manual
Ingens (OSDI’16)

- **Asynchronous allocation**
  - Huge pages allocated in the background

- **Utilization-threshold** based allocation
  - Tunable bloat vs. performance
  - Adaptive based on memory pressure

- **Fairness driven by per-process fairness metric**
  - Heuristic based on past behavior

---

- low latency
  - too many page faults

- manual

- weak correlation with page walk overhead
## Current state-of-the-art

<table>
<thead>
<tr>
<th></th>
<th>FreeBSD</th>
<th>Linux</th>
<th>Ingens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory bloat</td>
<td>Low</td>
<td>High</td>
<td>Tunable</td>
</tr>
<tr>
<td>Performance</td>
<td>Low</td>
<td>High</td>
<td>Tunable</td>
</tr>
<tr>
<td>Allocation latency</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td># page faults</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

- Hard to find the sweet-spot for utilization-threshold in Ingens
  - Application dependent, phase dependent
HawkEye
Key Optimizations

➢ Asynchronous page pre-zeroing\cite{1}
➢ Content deduplication based bloat mitigation
➢ Fine-grained intra-process allocation
➢ Fairness driven by hardware performance counters

\cite{1} Optimizing the Idle Task and Other MMU Tricks, OSDI'99
Asynchronous page pre-zeroing

- Pages zero-filled in the background

- Potential issues:
  - Cache pollution – leverage non-temporal writes
  - DRAM bandwidth consumption – rate-limited
    - Limit CPU utilization (e.g., 5%)
Asynchronous page pre-zeroing

Enables aggressive allocation with low latency

✓ 13.8x faster VM spin-up
✓ 1.26x higher throughput (Redis)
Mitigating bloat
Mitigating bloat

Virtual memory

Physical memory

huge page mapping
Mitigating bloat

Virtual memory

Physical memory

huge page mapping

unused
Mitigating bloat

Virtual memory

Physical memory

unused

tiny page mapping

huge page mapping

zero-filled
Mitigating bloat

- Observation: Unused base pages remain zero-filled
- Identify bloat by scanning memory
- Dedup zero-filled base pages to remove bloat
Mitigating bloat

- Ease of detecting non-zero pages

### Chart: Ease of detecting non-zero pages

<table>
<thead>
<tr>
<th>Offset (bytes)</th>
<th>CPU2006_int</th>
<th>CPU2006_fp</th>
<th>PARSEC</th>
<th>NPB</th>
<th>Graph500</th>
<th>PageRank</th>
<th>XSbench</th>
<th>Redis</th>
<th>Memcached</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>67.5</td>
<td>55.4</td>
<td>115.5</td>
<td>3.9</td>
<td>2.8</td>
<td>1.2</td>
<td>1</td>
<td>6.63</td>
<td>27.4</td>
<td>9.11</td>
</tr>
</tbody>
</table>
Mitigating bloat

✓ Automated "bloat vs. performance" management

Redis
P1: insert
P2: delete
P3: insert
<table>
<thead>
<tr>
<th>Tradeoff-1:</th>
<th>FreeBSD</th>
<th>Linux</th>
<th>Ingens</th>
<th>HawkEye</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory bloat</td>
<td>Low</td>
<td>High</td>
<td>Tunable</td>
<td>Automated</td>
</tr>
<tr>
<td>Performance</td>
<td>Low</td>
<td>High</td>
<td>Tunable</td>
<td>Automated</td>
</tr>
<tr>
<td>Allocation latency</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td># page faults</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>

**Tradeoff-2:**

- **FreeBSD**
- **Linux**
- **Ingens**
- **HawkEye**

- Memory bloat: Low
- Performance: Low
- Allocation latency: Low
- # page faults: High

**Tradeoff-1:**

- Memory bloat: Low
- Performance: Low
- Allocation latency: Low
- # page faults: High
Fine-grained (intra-process) allocation

- Maximizing performance with limited contiguity
Fine-grained (intra-process) allocation

- Maximizing performance with limited contiguity

access-coverage: # base pages accessed per second

- A good indicator of TLB-contention due to a region
Fine-grained (intra-process) allocation

- Track access-coverage (access_map)
- Allocate in the sorted order (top to bottom)
- Yields higher profit per allocation
Fine-grained (intra-process) allocation

Workload: XSBench
Fine-grained (intra-process) allocation

Execution time (ms) saved per huge page allocation

- **Graph500**
- **XSBench**
- **NPB_CG.D**

**Graph500**
- Linux: 900 ms
- Ingens: 0 ms
- HawkEye: 1200 ms

**XSBench**
- Linux: 0 ms
- Ingens: 0 ms
- HawkEye: 900 ms

**NPB_CG.D**
- Linux: 0 ms
- Ingens: 100 ms
- HawkEye: 200 ms
Fair (inter-process) allocation

- Prioritize allocation to the process with highest expected improvement

- How to estimate page walk overhead
  - Profile hardware performance counters
  - Low cost, accurate!
Fair (inter-process) allocation

Workloads running alongside a TLB-insensitive process
Summary

- OS support for huge pages involves complex tradeoffs
- Balancing fine-grained control with high performance
- Dealing with fragmentation for efficiency and fairness
Summary

- OS support for huge pages involves complex tradeoffs
- Balancing fine-grained control with high performance
- Dealing with fragmentation for efficiency and fairness

HawkEye: Resolving fundamental conflicts for huge page optimizations
https://github.com/apanwariisc/HawkEye
Thank You