Every Walk’s a Hit: Making Page Walks Single-Access Cache Hits

Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer
Today’s Virtual Memory
Translations don’t scale

- TLB reach is not scaling with DRAM
  - 8 MB/4 GB when using 4 KB/2 MB pages\(^1\)
- TLB misses resolved with Page Table Walks
  - Four serial memory accesses required
- Page Walk Caches help by skipping levels
  - \(1.1 \sim 2.5\) (avg 1.5) memory accesses per TLB miss
  - Workloads with memory sizes up to 8GB

Page Walker Cache coverages don’t scale!
E.g. Larger memory sizes, 5-level page tables

---
\(^1\) AMD Zen3 uArch
\(^2\) Van Schaik et al., Vrije Universiteit Tech report 2017. Coverage based on Intel Skylake uArch
Past Proposals: Require large contiguous physical memory

- Address Translation with Prefetching (ASAP)\textsuperscript{1} - Prefetch leaf nodes
  
  - Elastic Cuckoo Page Tables (ECPT)\textsuperscript{2} – multi-way hash PTs

---

\textsuperscript{1} Margaritov et al., MICRO '19
\textsuperscript{2} Skarlatos et al., ASPLOS '20
Challenge: Contiguous Physical Memory

• Page table allocation is a **critical task**
  • Cannot fail, cannot take too long (e.g. tail latency!)

• Difficult to get large physically contiguous memory allocations
  • Availability of contiguous memory is **not guaranteed**

• Implications: guarantee contiguous allocations
  or suffer latency spikes or PT allocation failures

Can we have a scalable page walk
that does not need contiguity guarantees?
Our Work

**Problem:** Page table walks $\Rightarrow$ 1.1-2.5 accesses (incl. DRAM), Large physically contiguous memory

**Solution:** Making Page Table Walks Single-access Cache Hits

- Flatten the page table
- Prioritize the page table in the cache
Conventional (4-level tree) Page Tables

Virtual Address: Virtual page number, Offset
Indices for each PT level

CR3/TTBR → L4 → L3 → Offset → Virtual address

Height of tree == Number of serial memory accesses:

\[ 1 + 2 + 3 + 4 \rightarrow 4 \text{ serial memory accesses} \]

* Worst case: Page walker cache helps reduce down to 1.5 - 2.5 in practice

Flatten the tree \(\rightarrow\) fatter nodes, shorter tree
Flattening two levels

Gather the scattered L1 Page tables of a L2 into one L2+L1 flattened page table

Flattened nodes stored in 2MB pages: leverages existing OS 2MB support
Flattening the Page Table

Flattening reduces serial memory accesses $4 \rightarrow 2$

Trade-off memory space for shorter page table walks
Various Combinations of Flattening

L4+L3 and L2+L1 Flattening
L3+L2 Flattening
L5+L4 and L3+L2 Flattening
L5+L4+L3 Flattening

Focus of this presentation
How does it work with 5-level PTs?
Many more combinations!
Different flattening options for different requirements
What if a 2MB Allocation Fails?

1. Allocating root node

2. Allocating L2+L1

Note: Flattened L2+L1 may coexist with unflattened L2 and L1 PTs

Fall back to allocating 4KB PTs

Specify next PT is 4KB
HW and SW Changes

Virtual Address | Virtual page number | Offset
---|---|---

Change index: PT increased 4 KB $\rightarrow$ 2MB

Index increases 9-bit $\rightarrow$ 18-bit

HW

- Virtual address indexing changes
- Pointers to PT need to specify size of PT

SW

- Allocate 2MB PTs
- Mark pointers with appropriate size
- Small changes (+614/-109 LOC)
Flattening and 2 MB data pages

Conventional PT and 2 MB Data

- L4 → L3 → L2 → L1 → 4 KB Data → 2 MB Data

Replicated 2MB entries

- L4+L3 → L2+L1 → 4 KB Data → 2 MB Data

Flattening + non-flattening

- L4+L3 → L2+L1 → 4 KB Data → 2 MB Data

- L4+L3 → 1GB VA → L2 → 2 MB Data

2 MB Data pointed by L2 PT entries

Flattened Entries can point to each sub-4 KB data page of 2 MB page

Requires more PT entries, bad cache performance

Don’t flatten 1 GB regions with many large pages!
Summary: Flattening

- Flattening reduces serial memory accesses
  - 4 → 2 memory accesses
  - In practice: 1.5 → 1 memory accesses (PWC)
- Memory access are still missing the cache
  - Long latency accesses to the DRAM

Can we do something about the DRAM accesses?
Prioritize Keeping Page Table in the Cache

• Workloads with high TLB miss show high data cache miss ratio
  • L2: 95% miss ratio, L3: 80% miss ratio

Data caching not effective
→ Cache page table instead!

Insight: PTE covers 64 cachelines
→ 64x as likely to see reuse

• Simple solution: bias replacement to keep page table entries
  • 99x more likely to evict data

GUPS* L3 Occupancy

<table>
<thead>
<tr>
<th></th>
<th>Data</th>
<th>PT</th>
</tr>
</thead>
<tbody>
<tr>
<td>59% Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

L3 Miss ratio

<table>
<thead>
<tr>
<th></th>
<th>Data</th>
<th>PT Walk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>99.1%</td>
<td>54.5%</td>
</tr>
<tr>
<td>PT Prioritization</td>
<td>99.5%</td>
<td>14.7%</td>
</tr>
</tbody>
</table>

Data miss ratio hardly changed, walker miss rate improved by 3.7x

* Workload that shows significant impact. Cherry-picked to emphasize behavior of prioritization
Evaluation Methodology

- Simulation infrastructure
  - Gem5
  - System-call emulation mode
- Graph, bioinformatics, SPEC CPU, linear classifiers, microbenchmarks

<table>
<thead>
<tr>
<th>Component</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>2GHz, OoO x86</td>
</tr>
<tr>
<td>Caches</td>
<td>L1 I/D: 32KB, 8-way</td>
</tr>
<tr>
<td></td>
<td>L2: 256KB, 8-way</td>
</tr>
<tr>
<td></td>
<td>L3: 16MB, 8-way</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR4-2400, 4 channels</td>
</tr>
<tr>
<td>L1 TLB</td>
<td>64-entry, 4-way, 4KB</td>
</tr>
<tr>
<td></td>
<td>32-entry, 4-way, 2MB</td>
</tr>
<tr>
<td>L2 TLB</td>
<td>1,536-entry, 12 way (Shared by 4KB and 2MB)</td>
</tr>
<tr>
<td>Page Structure</td>
<td>4-entry L4 cache entries</td>
</tr>
<tr>
<td>Cache</td>
<td>4-entry L3 cache entries</td>
</tr>
<tr>
<td></td>
<td>24-entry L2 cache entries</td>
</tr>
</tbody>
</table>
Performance Results

Outperforms prior work using smaller chunks (2MB) of contiguous memory
Looking further into the results

Memory requests per page walk

- Baseline
- Flattening

Page walk latency (cycles)

- Baseline
- Flattening
- Prioritizing
- Both

1.5 → 1.0 (avg)
Flattening alone (little benefit)
50.9 → 46.3 cycles (avg)
Prioritization
50.9 → 33.0 cycles (avg)
Both
50.9 → 29.1 cycles (avg)
Energy implications

ASAP issues prefetches for leaf page table entries → Increased cache accesses

ECPT issues multiple concurrent memory accesses, → Increased cache and memory accesses

Flattening reduces number of accesses per walk → Reduced cache accesses

Prioritization increases cache hits for walks → Reduced memory accesses

Flattening and Prioritization reduces both memory and cache accesses
More in the Paper

• Flattening
  • How we support large page data
  • Various flattening options
  • Flattening for virtualization
  • **Supporting recursive PTs (Windows)**

• Results
  • Multi-core results
  • Virtualization results
  • Performance results with different cache size
  • **Case study: flattening for mobile systems**
Conclusion

• Page table walks need to be short
  • Current page walk caching does not scale
  • Prior work require large contiguous physical memory and has no fallback path

• Flattening and Prioritizing → Single-access cache hitting page walks
  • Simple: Uses existing large pages for the page table
  • Practical: Provides graceful fallback when large pages are not available
  • Results: Performance gains 9.2% (14.0% in virtualized systems)
    Energy reductions (5.1% cache, 4.7% memory)
Every Walk’s a Hit: Making Page Walks Single-Access Cache Hits

Chang.Hyun.Park@it.uu.se, Ilias.Vougioukas@arm.com, Andreas.Sandberg@arm.com, and David.Black-Schaffer@it.uu.se