Page Table Walk Aware Cache Management for Efficient Big Data Processing

Eishi Arima†‡ and Hiroshi Nakamura†

†The University of Tokyo
‡Lawrence Livermore National Laboratory
Executive Summary

• **Background**: The performance penalty of *page table walks* after TLB misses is serious for modern computer systems.
  - It is reported around 40% of the total execution time is spent on the virtual to physical address translation while executing applications that use *very large memory with irregular access patterns* [A.Bhattacharjee+MICRO2013]

• **Our focus**: *Cache management* for mitigating the overhead of page table walks
  - Caches are accessed during page table walks to fetch *Page Table Entries (PTEs)* but not optimized accordingly

• **Our proposal**: Storing PTEs preferentially on *upper level caches*
  - *PTEs* fundamentally have *much higher locality* than usual data
  - *PTEs* are usually *evicted* from upper level caches before re-referenced due to the conflict with data brought by frequent cache misses
Virtual memory and page table walk

- Virtual memory is ubiquitous in modern computer systems
  - Benefits: process isolation, inter-process data sharing and memory capacity management

- Page table: used for virtual to physical memory translation
  - In modern systems, it is implemented with a 4-level radix tree
    - A table access called page table walk requires 4 times memory references
Conventional hardware for accelerating page table walk

- **TLB**: A cache that keeps page table entries (PTEs, the lowest level entries of page tables)
  - Due to the size limitation, only few 10s of PTEs are cached on a TLB
- **Page walker cache (PWC)**: A cache that keeps higher level entries of page tables
  - Also known as MMU Cache
- In addition, *usual data caches* can also keep these entries
Performance overhead of page table walks

- The overhead of page table walks accounts for 43% of total execution time at the worst case.
- 28% of the execution time is spent on PTE accesses which follow after PWC accesses.
  - *We should revisit the cache management for PTEs to mitigate the performance overhead of page table walks.*
Goal, observation and proposal

• **Goal:**
  • Mitigating the performance overhead of page table walks by optimizing allocations of PTEs on caches

• **Observations:**
  • PTEs fundamentally have much higher locality than usual data
  • PTEs are usually evicted from upper level caches before re-referenced

• **Proposal:**
  • Cache replacement algorithm that preferentially stores PTEs on caches
Goal, observation and proposal

- **Goal:**
  - Mitigating the performance overhead of page table walks by optimizing allocations of PTEs on caches

- **Observations:**
  - PTEs fundamentally have much higher locality than usual data
  - PTEs are usually evicted from upper level caches before re-referenced

- **Proposal:**
  - Cache replacement algorithm that preferentially stores PTEs on caches
Access locality difference between PTEs and data

- A PTE must be referenced to access data.
- A PTE is associated with a page (4KB), thus the number of accesses to it is the same as that to the page.
- So, a PTE is accessed much more times than one of data in the associated page even though they occupy the same size (8Byte).
  - Thus, PTEs should remain in upper level caches

![Diagram showing access locality difference between PTEs and data]
Hit rate of PTE accesses

- Although PTE accesses have higher locality, they cause frequent misses on upper level caches
  - In the worst case, the PTE hit rate on L1D cache is only 15%
Conflict on upper level caches

• This is because PTEs are evicted from L1D cache due to conflict with large data brought by frequent misses
  • APKI of PTEs < MPKI of data, on L1D cache
    • APKI: Access Per Kilo Instruction
    • MPKI: Miss Per Kilo Instruction

• We need to prevent PTEs from eviction!
Goal, observation and proposal

• **Goal:**
  • Mitigating the performance overhead of page table walks by optimizing allocations of PTEs on caches

• **Observations:**
  • PTEs fundamentally have much higher locality than usual data
  • PTEs are usually evicted from upper level caches before re-referenced

• **Proposal:**
  • Cache replacement algorithm that preferentially stores PTEs on caches
Concept

- Classifying cache lines as follows
  - **PTEs, Hot data** and **Cold data**
    - **Hot data**: intensively accessed data
    - **Cold data**: non-reused dead data or reused very far future
  - Goal: keeping **PTEs** on caches without evicting **hot data** from them

**Conventional**

```
<table>
<thead>
<tr>
<th>PTEs</th>
<th>hot data</th>
<th>cold data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

evicted

later
```

**Goal**

```
<table>
<thead>
<tr>
<th>PTEs</th>
<th>hot data</th>
<th>cold data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

evicted

later
```

4/9/17 BPOE-8
Basic replacement algorithm

• Our algorithm has a stack to define *eviction priority* like LRU
• Our algorithm consists of *insertion/promotion* policies which use the stack like LRU
  • *insertion*: defining the eviction order of incoming line
  • *promotion*: updating the eviction order for reused line
• The figure below shows the example of LRU

![Example of LRU](image-url)
Proposed replacement algorithm

- Our algorithm considers the classification of **PTEs**, **hot data** and **cold data** in **Insertion/Promotion** policies unlike LRU
  - **Insertion:** The eviction priority of incoming **PTE line** is set to 0, but a **data line** is set to 1
    - Most of the data lines keep only non-reused **cold data**
  - **Promotion:** The eviction priority of reused line is set to 0
    - We regard re-referenced data in the cache as **hot data (or PTEs)**

![Eviction Priority Diagram](image)
Evaluation setup

• We evaluated our method with full system simulator Gem5
• We selected several applications from SPEC2006, Biobench and Parsec, which require quite large memory with irregular access patterns
• The Insertion Position \( I \) is optimized for each cache so that the geometric mean of performance is maximized
  • So, \( I \) is set the same value for all applications

<table>
<thead>
<tr>
<th>Name</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>1 core, 2GHz, x86, OoO, 4-way fetch/decode/issue</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.6.28, 4KB page</td>
</tr>
<tr>
<td>L1 D/I cache</td>
<td>32KB, 1-cycle latency, 8-way set assoc., 64B line</td>
</tr>
<tr>
<td>L1 D/I TLB</td>
<td>64-entry, 1-cycle latency, full assoc.</td>
</tr>
<tr>
<td>page walk cache</td>
<td>256-entry, 2-cycle latency, 8-way set assoc.</td>
</tr>
<tr>
<td>(L4/L3/L2 unified)</td>
<td></td>
</tr>
<tr>
<td>(private) L2 cache</td>
<td>256KB, 12-cycle latency 8-way set assoc., 64B line</td>
</tr>
<tr>
<td>(D/I unified)</td>
<td></td>
</tr>
<tr>
<td>(shared) L3 cache</td>
<td>2MB, 30-cycle latency 16-way set assoc., 64B line</td>
</tr>
<tr>
<td>Main memory</td>
<td>200-cycle latency</td>
</tr>
</tbody>
</table>
Performance comparison

• Proposed method improves performance few % compared to the conventional LRU
  • Proposed method outperform LRU for all applications
  • But, it still has room for improvement as shown in later slides
PTE hit rate on L1d cache

- Because L1D cache hit rate is the most important, we focus on its result in this presentation.
- Our proposal improves *PTE hit rate* on L1d cache about 10% on average and 40% at the best.
- But, there is still large *room for improvement*.
  - So, more aggressive method is necessary (future work).

![Room for improvement chart]

<table>
<thead>
<tr>
<th>Application</th>
<th>LRU</th>
<th>Proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>omnetpp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>streamcluster</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>canmeal</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>soplex</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>nummer</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>mcg</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>astar</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>GemFDTD</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>tiger</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>Avg.</td>
<td>10%</td>
<td></td>
</tr>
</tbody>
</table>
Data hit rate on L1d cache

• Our proposal does not decrease the data hit rate on L1D cache
  • Proposed method can successfully keep intensively accessed *hot data* on L1d cache, and only non-reused *cold data* are replaced by PTEs
Conclusion and future direction

• Conclusion:
  • To mitigate the overhead of page table walks, we proposed and evaluated a cache management scheme that optimize the allocation of PTEs on caches.

• Future direction:
  • Developing more sophisticated method that can more aggressively store PTEs on upper level caches
    • The insertion positions $I$ of should be dynamically optimized during program execution time
    • Software-side approach may necessary
  • Evaluating our methods with well-known big data workloads like BigDataBench[L. Wang+HPCA2014]