DAAIP: Deadblock Aware Adaptive Insertion Policy for High Performance Caching

Newton, Sujit Kr Mahto, Suhit Pai and Virendra Singh

Computer Architecture and Dependable Systems Laboratory
Department of Electrical Engineering
Indian Institute of Technology Bombay, India

The 35th IEEE International Conference on Computer Design
Outline

1. Background
2. Motivation
3. Deadblock Aware Adaptive LLC Replacement Policy
4. Conclusion
Background

- Fast processor + Slow memory ⇒ Cache hierarchy

- Multicore System: Uses 3 levels of cache hierarchy, each level to meet different requirements
  - L1 Cache
    - Smallest (32 KB), Fastest (*load-to-use* latency: 1-3 cycles\(^1\))
    - Miss penalty can be hidden by utilizing the ILP present
  - L2 Cache
    - Small (256 KB), Fast (latency: 14-15 cycles)
    - On miss, core *stalls-on-use*
  - L3 Cache (generally Last-level Cache (LLC))
    - Large (8 MB), Slow (latency: 30-40 cycles)
    - Miss penalty of an LLC miss ~ 200 cycles, cannot be hidden

- Hence, it's important to reduce LLC misses

---
\(^1\)Caches’ size and their latency data is for *Nehalem* Processor

Newton (IIT Bombay)  DAAIP  Tuesday, 7\(^{th}\) Novembers, 2017  4 / 32
Outline

1. Background
2. Motivation
3. Deadblock Aware Adaptive LLC Replacement Policy
4. Conclusion
Zero Reuse Cache Lines

Insertion

Demand
Hit

Eviction

Re-Ref 1

Re-Ref n
No further Re-Ref

Dead
Zero Reuse Cache Lines

Insertion

Eviction

Zero Reuse Line: Deadblock

Re-Ref 1

Re-Ref n

No further Re-Ref

Dead

Demand Hit

No Re-Ref between insertion and eviction

 demand

Hit

Hit

Demand

DAAIP

Tuesday, 7th Novembers, 2017

7 / 32
Insights:

- Benchmarks like `lbm`, `libquantum`, `milc` have almost 100% deadblocks
Zero Reuse Cache Lines in LLC

Insights:

- Benchmarks like *lbm*, *libquantum*, *milc* have almost 100% deadblocks.
- On an average 86% of cache lines are deadblocks.
Zero Reuse Cache Lines in LLC

Reasons

- LLC observes filtered temporal and spatial locality

Graph showing the percentage of dead blocks for various benchmarks and the average.
Zero Reuse Cache Lines in LLC

Reasons

- LLC observes filtered temporal and spatial locality
- Blocks with long *reuse distance* get thrashed
Cache Behaviour of Programs

Misses per kilo instruction (MPKI)

Ways allocated from 8-way 2MB LRU managed L3 cache

bzip
gcc
milc
libquantum

Newton (IIT Bombay)
DAAIP
Tuesday, 7th Novembers, 2017 10 / 32
Cache Behaviour of Programs

Negative Interference

*bzip* has MPKI of 5.7 when *bzip-libquantum* share LLC
Problem: LRU replacement is inefficient for LLCs
**Problem:** LRU replacement is inefficient for LLCs

**Goal:** A replacement policy that has following features:

- Low hardware overhead
- Low complexity
- High performance

Proposal: A mechanism to quickly separate out deadblocks and liveblocks and also, quickly evict deadblocks from LLC.
Overview of LLC Management

- **Problem**: LRU replacement is inefficient for LLCs

- **Goal**: A replacement policy that has following features:
  - Low hardware overhead
**Problem**: LRU replacement is inefficient for LLCs

**Goal**: A replacement policy that has following features:
- Low hardware overhead
- Low complexity
Overview of LLC Management

- **Problem**: LRU replacement is inefficient for LLCs

- **Goal**: A replacement policy that has following features:
  - Low hardware overhead
  - Low complexity
  - High performance

Proposal: A mechanism to quickly separate out deadblocks and liveblocks and also, quickly evict deadblocks from LLC.
Overview of LLC Management

- **Problem:** LRU replacement is inefficient for LLCs

- **Goal:** A replacement policy that has following features:
  - Low hardware overhead
  - Low complexity
  - High performance

- **Proposal:** A mechanism to quickly separate out *deadblocks* and *liveblocks* and also, quickly evict *deadblocks* from LLC
Replacement Policies in LLC

Replacement Policies in LLC

MRU 0 1 N-Way Cache Set N-2 N-1 LRU

LRU Insertion Policy

1 M. K. Qureshi et al. “Adaptive insertion policies for high performance caching”, ISCA’07
2 Aamer Jaleel et al. “High performance cache replacement using re-reference interval prediction (RRIP)”, ISCA’10
Replacement Policies in LLC

1M. K. Qureshi et al. “Adaptive insertion policies for high performance caching”, ISCA’07
2Aamer Jaleel et al. “High performance cache replacement using re-reference interval prediction (RRIP)”, ISCA’10
Replacement Policies in LLC

Non-adaptive
- Across Benchmarks

MRU  0  1  N-Way Cache Set  N-2  N-1  LRU

RRIP Policy

Replacement Policies in LLC

Non-adaptive
- Across Benchmarks
- Within Benchmarks

RRIP Policy

MRU  0  1  N-Way Cache Set  N-2  N-1  LRU

1 M. K. Qureshi et al. “Adaptive insertion policies for high performance caching”, ISCA’07
2 Aamer Jaleel et al. “High performance cache replacement using re-reference interval prediction (RRIP)”, ISCA’10
Outline

1. Background
2. Motivation
3. Deadblock Aware Adaptive LLC Replacement Policy
4. Conclusion
Phase-wise Behaviour Among Benchmarks

Figure: *bzip*
Figure: milc
Phase-wise Behaviour Among Benchmarks

Adaptive Policy
Deadblock Aware Adaptive LLC Replacement Policy

(a) bzip

(b) milc
Deadblock Aware Adaptive LLC Replacement Policy

Figure: Architecture of DAAIP
Deadblock Aware Adaptive LLC Replacement Policy

1: **ForEachCacheMiss:**
2: \[\text{if } \text{NewInsertion}Ctr = \text{PhaseLen} \text{ then}\]
3: \[\text{Compute percentageOfDeadblocks}\]
4: \[\text{if } \text{percentageOfDeadblocks} > \text{Threshold} \text{ then}\]
5: \[\text{Insert new blocks at LRU}\]
6: \[\text{else}\]
7: \[\text{Insert new blocks at (LRU - 1)}\]
8: \[\text{end if}\]
9: \[\text{NewInsertion}Ctr = \text{NewInsertion}Ctr / 2\]
10: \[\text{DeadBlock}Ctr = \text{DeadBlock}Ctr / 2\]
11: \[\text{else}\]
12: \[\text{NewInsertion}Ctr++\]
13: \[\text{if } \text{BlockEvicted}Is\text{Dead} \text{ then}\]
14: \[\text{DeadBlock}Ctr++\]
15: \[\text{end if}\]
16: \[\text{end if}\]
## Simulation Framework

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1-D Cache</strong></td>
<td>32 KB, 4-Way, LRU, Private, 4 cycles</td>
</tr>
<tr>
<td><strong>L1-I Cache</strong></td>
<td>32 KB, 4-Way, LRU, Private, 4 cycles</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td>256 KB, 8-Way, LRU, Private, 8 cycles</td>
</tr>
<tr>
<td><strong>L3 Cache</strong></td>
<td>2MB per-core, 16-way, Shared, 30 cycles</td>
</tr>
<tr>
<td><strong>Main Memory Latency</strong></td>
<td>175 Cycles</td>
</tr>
<tr>
<td><strong>Baseline Processor</strong></td>
<td>x86 Nehalem microarchitecture, 2.67 GHz, 4-wide fetch, 128-entry ROB</td>
</tr>
</tbody>
</table>
DAAIP: Individual MPKI Reduction over SRRIP

Newton (IIT Bombay)
Performance gains over SRRIP: 5.8%

---

1 Aamer Jaleel et al. “High performance cache replacement using re-reference interval prediction (RRIP)” ISCA'10
Performance gains over ABRIP: 4.6%

1 Parth Lathigara et al. “Application behavior aware re-reference interval prediction for shared LLC”, ICCD’15
DAAIP: Impact of Varying Thresholds

Minimal gains over chosen 16 bit Counters and 90% Deadblock threshold
### Hardware Overhead of DAAIP over SRRIP

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit per tag-store entry to identify the core</td>
<td>1 b</td>
</tr>
<tr>
<td>1 bit per block to track if it is dead at the time of eviction</td>
<td>1 b</td>
</tr>
<tr>
<td>Number of cache blocks in 4MB LLC with 64 B blocksize</td>
<td>62500</td>
</tr>
<tr>
<td>Overhead per block (62500*2) bits</td>
<td>15625 B</td>
</tr>
<tr>
<td>4 Counters each of size 16 bits</td>
<td>8 B</td>
</tr>
<tr>
<td>Total Hardware Overhead</td>
<td>15633 B</td>
</tr>
<tr>
<td>Percentage increase over 4MB LLC area due to DAAIP</td>
<td>0.0039 %</td>
</tr>
</tbody>
</table>
1. Background
2. Motivation
3. Deadblock Aware Adaptive LLC Replacement Policy
4. Conclusion
Conclusions

- LRU is inefficient for L3 caches. Most lines remain unused between insertion and eviction.
Conclusions

- LRU is inefficient for L3 caches. Most lines remain unused between insertion and eviction.

- Proposed changes to cache insertion policy (DAAIP) has:

  - Negligible hardware overhead: (0.0039% over LLC)
  - Low complexity: Straight-forward to implement
  - Significant performance improvement: $\sim 6\%$ throughput gain
Conclusions

• LRU is inefficient for L3 caches. Most lines remain unused between insertion and eviction.

• Proposed changes to cache insertion policy (DAAIP) has:
  • Negligible hardware overhead: (0.0039% over LLC)
Conclusions

- LRU is inefficient for L3 caches. Most lines remain unused between insertion and eviction.

- Proposed changes to cache insertion policy (DAAIP) has:
  - Negligible hardware overhead: (0.0039% over LLC)
  - Low complexity: Straight-forward to implement
Conclusions

- LRU is inefficient for L3 caches. Most lines remain unused between insertion and eviction.

- Proposed changes to cache insertion policy (DAAIP) has:
  - Negligible hardware overhead: (0.0039% over LLC)
  - Low complexity: Straight-forward to implement
  - Significant performance improvement: ~ 6% throughput gain
Namaste! Thanks!
newton@ee.iitb.ac.in
Better to allocate LLC to *cache-friendly* benchmarks like *bzip, soplex*
## Benchmarks: APKI & MPKI

<table>
<thead>
<tr>
<th>Application</th>
<th>apki</th>
<th>mpki</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip</td>
<td>15.50</td>
<td>5.11</td>
<td>Cache Friendly (Cf)</td>
</tr>
<tr>
<td>soplex</td>
<td>38.30</td>
<td>19.15</td>
<td>Cache Friendly (Cf)</td>
</tr>
<tr>
<td>gcc</td>
<td>28.38</td>
<td>7.93</td>
<td>Cache Friendly (Cf)</td>
</tr>
<tr>
<td>sphinx</td>
<td>13.45</td>
<td>11.78</td>
<td>Cache Friendly (Cf)</td>
</tr>
<tr>
<td>libquantum</td>
<td>49.45</td>
<td>35.39</td>
<td>Streaming (Str)</td>
</tr>
<tr>
<td>lbm</td>
<td>31.92</td>
<td>31.57</td>
<td>Streaming (Str)</td>
</tr>
<tr>
<td>milc</td>
<td>26.43</td>
<td>23.68</td>
<td>Streaming (Str)</td>
</tr>
<tr>
<td>cactusADM</td>
<td>4.89</td>
<td>5.20</td>
<td>Cache Friendly (Cf)</td>
</tr>
<tr>
<td>GemsFDTD</td>
<td>20.52</td>
<td>28.34</td>
<td>Cache Friendly (Cf)</td>
</tr>
<tr>
<td>hmmer</td>
<td>1.31</td>
<td>3.07</td>
<td>Cache Friendly (Cf)</td>
</tr>
<tr>
<td>leslie</td>
<td>21.67</td>
<td>23.80</td>
<td>Cache Friendly (Cf)</td>
</tr>
<tr>
<td>perlbench</td>
<td>0.94</td>
<td>1.34</td>
<td>Cache Friendly (Cf)</td>
</tr>
<tr>
<td>zeusmp</td>
<td>3.70</td>
<td>5.42</td>
<td>Cache Friendly (Cf)</td>
</tr>
</tbody>
</table>