Improving Cache Performance by Exploiting Read-Write Disparity

Samira Khan, Alaa R. Alameldeen, Chris Wilkerson, Onur Mutlu, and Daniel A. Jiménez
Summary

• Read misses are more critical than write misses
  • Read misses can stall processor, writes are not on the critical path

• **Problem:** Cache management does not exploit read-write disparity

• **Goal:** Design a cache that favors reads over writes to improve performance
  • Lines that are *only written to* are less critical
  • Prioritize lines that service *read requests*

• **Key observation:** Applications differ in their read reuse behavior in clean and dirty lines

• **Idea:** Read-Write Partitioning
  • Dynamically partition the cache between clean and dirty lines
  • Protect the partition that has more read hits

• **Improves performance over three recent mechanisms**
Outline

• Motivation
• Reuse Behavior of Dirty Lines
• Read-Write Partitioning
• Results
• Conclusion
Motivation

• Read and write misses are not equally critical
• Read misses are more critical than write misses
  • Read misses can stall the processor
  • Writes are not on the critical path

Cache management does not exploit the disparity between read-write requests
Key Idea

- Favor reads over writes in cache
- Differentiate between read vs. only written to lines
- Cache should protect lines that serve read requests
  - Lines that are only written to are less critical
- Improve performance by maximizing read hits
- An Example

![Diagram](image)
An Example

LRU Replacement Policy

Read-Biased Replacement Policy

Evicting lines that are only written to can improve performance by saved cycles depending on read requests.

2 stalls per iteration

1 stall per iteration
Outline

• Motivation
• Reuse Behavior of Dirty Lines
• Read-Write Partitioning
• Results
• Conclusion
Reuse Behavior of Dirty Lines

• Not all dirty lines are the same
• Write-only Lines
  • Do not receive read requests, can be evicted
• Read-Write Lines
  • Receive read requests, should be kept in the cache

Evicting write-only lines provides more space for read lines and can improve performance
Applications have different read and write behavior. 

- On average, 37.4% lines are write-only, 9.4% lines are both read and written.
Outline

• Motivation
• Reuse Behavior of Dirty Lines
• **Read-Write Partitioning**
• Results
• Conclusion
Read-Write Partitioning

• **Goal:** Exploit different read reuse behavior in dirty lines to maximize number of read hits

• **Observation:**
  – Some applications have more reads to clean lines
  – Other applications have more reads to dirty lines

• **Read-Write Partitioning:**
  – Dynamically partitions the cache in clean and dirty lines
  – Evict lines from the partition that has less read reuse

**Improves performance by protecting lines with more read reuse**
Applications have significantly different read reuse behavior in clean and dirty lines.

Read-Write Partitioning

**Graphs**

- **Soplex**
  - Clean Line: Blue triangles
  - Dirty Line: Red squares

- **Xalanc**
  - Clean Line: Blue triangles
  - Dirty Line: Red squares

**Graph Details**

- **X-axis**: Instructions (M)
- **Y-axis**: Number of Reads Normalized to Reads in clean lines at 100m

**Legend**

- Clean Line
- Dirty Line
Read-Write Partitioning

- Utilize disparity in read reuse in clean and dirty lines
- Partition the cache into clean and dirty lines
- Predict the partition size that maximizes read hits
- Maintain the partition through replacement
  - DIP [Qureshi et al. 2007] selects victim within the partition
Predicting Partition Size

- Predicts partition size using sampled shadow tags
  - Based on utility-based partitioning [Qureshi et al. 2006]
- Counts the number of read hits in clean and dirty lines
- Picks the partition \((x, \text{associativity} – x)\) that maximizes number of read hits

**Diagram:**

- **Counters**
  - Maximum number of read hits
- **Sets**
  - SAM P L E D
  - Clean
  - Dirty
  - MRU
  - MRU-1
  - LRU
  - LRU+1
  - TAG S

---

**Note:** The diagram illustrates the counting and selection process for clean and dirty lines, emphasizing the maximization of read hits through utility-based partitioning.
Outline

• Motivation
• Reuse Behavior of Dirty Lines
• Read-Write Partitioning
• Results
• Conclusion
Methodology

- CMP$im x86 cycle-accurate simulator [Jaleel et al. 2008]
- 4MB 16-way set-associative LLC
- 32KB I+D L1, 256KB L2
- 200-cycle DRAM access time
- 550m representative instructions
- Benchmarks:
  - 10 memory-intensive SPEC benchmarks
  - 35 multi-programmed applications
Comparison Points

• **DIP, RRIP: Insertion Policy** [Qureshi et al. 2007, Jaleel et al. 2010]
  – Avoid thrashing and cache pollution
    • Dynamically insert lines at different stack positions
  – **Low overhead**
  – **Do not differentiate between read-write accesses**

• **SUP+: Single-Use Reference Predictor** [Piquet et al. 2007]
  – Avoids cache pollution
    • Bypasses lines that do not receive re-references
  – **High accuracy**
  – **Does not differentiate between read-write accesses**
    • Does not bypass write-only lines
  – **High storage overhead, needs PC in LLC**
Comparison Points: Read Reference Predictor (RRP)

• A new predictor inspired by prior works [Tyson et al. 1995, Piquet et al. 2007]

• Identifies read and write-only lines by allocating PC
  – Bypasses write-only lines

• Writebacks are not associated with any PC

<table>
<thead>
<tr>
<th>PC P: Rd A</th>
<th>Wb A</th>
<th>Wb A</th>
</tr>
</thead>
</table>

| PC Q: Wb A | Wb A | Wb A |

Allocating PC from L1
Differencing read only vs. write-only lines improves performance over recent mechanisms.

<table>
<thead>
<tr>
<th></th>
<th>DIP</th>
<th>RRIP</th>
<th>SUP+</th>
<th>RRP</th>
<th>RWP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup vs. LRU</td>
<td>1.00</td>
<td>1.05</td>
<td>1.10</td>
<td>1.15</td>
<td>1.20</td>
</tr>
</tbody>
</table>

- DIP: 48.4KB
- RRIP: 2.6KB
- RWP performs within 3.4% of RRP, but requires 18X less storage overhead.

Differencing improves performance by 18X less storage overhead.
4 Core Performance

Differencing read vs. write-only lines improves performance over recent mechanisms.

More benefit when more applications are memory intensive.

Diagram showing speedup vs. baseline LRU for different memory intensities and mechanisms.
Average Memory Traffic

Increases writeback traffic by 2.5%, but reduces overall memory traffic by 16%
Partition size varies significantly for some benchmarks
Dirty Partition Sizes

Partition size varies significantly during the runtime for some benchmarks
Outline

• Motivation
• Reuse Behavior of Dirty Lines
• Read-Write Partitioning
• Results
• Conclusion
Conclusion

- **Problem:** Cache management does not exploit read-write disparity

- **Goal:** Design a cache that favors read requests over write requests to improve performance
  - Lines that are *only written to* are *less critical*
  - **Protect** lines that serve *read requests*

- **Key observation:** Applications differ in their read reuse behavior in clean and dirty lines

- **Idea:** Read-Write Partitioning
  - Dynamically partition the cache in clean and dirty lines
  - Protect the partition that has more read hits

- **Results:** Improves performance over three recent mechanisms
Thank you
Improving Cache Performance by Exploiting Read-Write Disparity

Samira Khan, Alaa R. Alameldeen, Chris Wilkerson, Onur Mutlu, and Daniel A. Jiménez
Extra Slides
Reuse Behavior of Dirty Lines in LLC

• Different read reuse behavior in dirty lines
  
  • **Read Intensive/Non-Write Intensive**
    – Most accesses are reads, only a few writes
    – Example: 483.xalancbmk
  
  • **Write Intensive**
    – Generates huge amount of intermediate data
    – Example: 456.hmmer
  
  • **Read-Write Intensive**
    – Iteratively reads and writes huge amount of data
    – Example: 450.soplex
Read Intensive/Non-Write Intensive

- Most accesses are reads, only a few writes
- 483.xalancbmk: *Extensible stylesheet language transformations (XSLT)* processor

- 92% accesses are reads
- 99% write accesses are stack operation
Non-Write Intensive

```

Non-Write Intensive

1.8% lines in LLC are write-only
These dirty lines should be evicted
```
Write Intensive

• Generates huge amount of intermediate data
• 456.hmmer: Searches a protein sequence database

- Hidden Markov Model of Multiple Sequence Alignments
- Database
- Best Ranked Matches

- Viterbi algorithm, uses dynamic programming
- Only 0.4% writes are from stack operations
Write Intensive

92% lines in LLC are write-only
These lines can be evicted
Read-Write Intensive

• Iteratively reads and writes huge amount of data
• 450.soplex: Linear programming solver

- Inequalities and objective function

- Simplex algorithm, iterates over a matrix
- Different operations over the entire matrix at each iteration
19% lines in LLC write-only, 10% lines are read-written.
Read and written lines should be protected.
Read Reuse of Clean-Dirty Lines in LLC

On average 37.4% blocks are dirty non-read and 42% blocks are clean non-read
Single Core Performance

5% speedup over the whole SPECCPU2006 benchmarks

- DIP
- RRIP
- SUP+
- RRP
- RWP

Speedup vs. Baseline LRU

48.4KB vs. 2.6KB
On average 14.6% speedup over baseline
5% speedup over the whole SPECCPU2006 benchmarks
Writeback Traffic to Memory

Increases traffic by 17% over the whole SPECCPU2006 benchmarks
4-Core Performance

<table>
<thead>
<tr>
<th>Workloads</th>
<th>RRP</th>
<th>RWP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Normalized Weighted Speedup

Workloads
Change of IPC with Static Partitioning

**Xalancbmk**

**Soplex**

<table>
<thead>
<tr>
<th>Number of Ways Assigned to Dirty Blocks</th>
<th>IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.9</td>
</tr>
<tr>
<td>2</td>
<td>0.8</td>
</tr>
<tr>
<td>4</td>
<td>0.7</td>
</tr>
<tr>
<td>6</td>
<td>0.6</td>
</tr>
<tr>
<td>8</td>
<td>0.5</td>
</tr>
<tr>
<td>10</td>
<td>0.4</td>
</tr>
<tr>
<td>12</td>
<td>0.3</td>
</tr>
<tr>
<td>14</td>
<td>0.2</td>
</tr>
<tr>
<td>16</td>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Ways Assigned to Dirty Blocks</th>
<th>IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>0.7</td>
</tr>
<tr>
<td>4</td>
<td>0.6</td>
</tr>
<tr>
<td>6</td>
<td>0.5</td>
</tr>
<tr>
<td>8</td>
<td>0.4</td>
</tr>
<tr>
<td>10</td>
<td>0.3</td>
</tr>
<tr>
<td>12</td>
<td>0.2</td>
</tr>
<tr>
<td>14</td>
<td>0.1</td>
</tr>
<tr>
<td>16</td>
<td>0.0</td>
</tr>
</tbody>
</table>