Multi-Core Memory Hierarchies

Lecture 3: LLC Replacement and Bypassing

Tong Dong

http://mprc.pku.edu.cn/~tongdong/MMH
Model of cache performance

For N cores and T thread per core, the miss ratio of the $i$-level cache:

$$m_L_i = C_L_i N T \beta_{L_i}$$

- $\alpha, \beta$: locality(Replacement) function constant, $>1$, $>>1$
- $I$: interference function constant, $>1$
- $C$: cache size in level $i$

Average Access Time in $i$-level cache:

$$T_{L_i} = 1 - m_{L_i} T_{L_i} \text{hit} + m_{L_i} T_{L_i}^{-1}$$

- $T$: Access Time
Basic Replacement Policies

- **LRU**: least recently used

- **LFU**: least frequently used (requires small saturating counters)

- **pseudo-LRU**: organize ways as a tree and track which sub-tree was last accessed

- **NRU**: every block has a bit; the bit is reset to 0 upon touch; when evicting, pick a block with its bit set to 1; if no block has a 1, make every bit 1
Why the Basic Policies Fail

Access types that pollute the cache without yielding too many hits: streaming (no reuse), thrashing (distant reuse)

Current hit rates are far short of those with an oracular replacement policy (Belady): evict the block whose next access is most distant

A large fraction of the cache is useless – blocks that have serviced their last hit and are on the slow walk from MRU to LRU
Cache Replacement Policy

Inclusive vs Exclusive LLC

Primary Replacement policies
- Victim Selection
- Block Insertion
- Block Promotion

Bypassing
- Non-Insertion
Cost-based Replacement
- Low-cost miss if high memory-level parallelism (MLP)
- MHSR counters per entry
- Linear combination of recency and MLP-cost

LIN (Linear) Policy

Set-Sampling: Sampling Based Adaptive (SBAR)
- Tournament Selection (TSEL)
More than half cache lines is Zero Reuse Lines

Replacement policy
– victim selection policy and insertion policy
Common Cache Access Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Sequence Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recency-Friendly</td>
<td>((a_1, \ldots, a_{k-1}, a_k, a_k, a_{k-1}, \ldots, a_1)^N)</td>
</tr>
<tr>
<td>Thrashing ((k &gt; C))</td>
<td>((a_1, \ldots, a_{k-1}, a_k)^N)</td>
</tr>
<tr>
<td>Streaming ((k = \infty))</td>
<td>((a_1, \ldots, a_{k-1}, a_k)^N)</td>
</tr>
<tr>
<td>Mixed ((k &lt; C, m &gt; C))</td>
<td>([((a_1, \ldots, a_k)^A P_{\epsilon}(b_1, \ldots, b_m))^N]</td>
</tr>
</tbody>
</table>

- \(C\) represents the cache set associativity.
- \(a_i\) represents a cache line access.
- \((a_1, a_2, \ldots, a_{k-1}, a_k)\) is a temporal sequence of \(k\) unique addresses to a cache set.
- \((a_1, a_2, \ldots, a_{k-1}, a_k)^N\) represents a temporal sequence that repeats \(N\) times.
- \(P_{\epsilon}(a_1, a_2, \ldots, a_{k-1}, a_k)\) is a temporal sequence that occurs with some probability \(P_{\epsilon}\).
Optimal Replacement Policy?

[Belady, IBM Systems Journal, 1966]
Evict block with longest reuse distance
– i.e. next reference to block is farthest in future
– Requires knowledge of the future!
Can’t build it, but can model it with trace
– Process trace in reverse
– LRU stack algorithm: How to do this in one pass over the trace with some lookahead (Cheetah simulator)
Useful, since it reveals opportunity
– (X,A,B,C,D,X): LRU 4-way SA $, 2\textsuperscript{nd} X will miss
DIP: Dynamic Insertion Policy

LRU Insertion Policy (LIP)
- Thrashing
- Places coming line in LRU position instead of MRU
- Bypassing for not inclusive cache

Bimodal Insertion Policy (BIP)
- Infrequently places some incoming lines into MRU

Dynamic Insertion Policy (DIP)
- Dynamically estimates the number of misses incurred by the two competing insertion policies and selects the policy that incurs the fewest misses.

Reduces MPKI 21% and 2/3 between LRU/OPT

Set Dueling

**Pseudo-LIFO, Chaudburi, MICRO’09**

Fill Stack: all blocks in the order in arrival time. Most cache hits incurs at the top of the fill stack. Blocks at the top of the fill stack should be prioritized for victimization.
Software-based page coloring scheme
Isolate pages with high miss rates to a small region of the LLC.
Re-Reference Interval Prediction: in essence, insert blocks near the end of the list than at the very end

Implement with a multi-bit version of NRU: zero counter on touch, evict block with max counter, else increment every counter by one

RRIP can be easily implemented by setting the initial counter value to max-1 (does not require list management)
RRIP, Jallel et al., ISCA’10

Re-use distance vs. Re-reference distance
Re-Reference Interval Prediction (RRIP)
Not Recently Used (NRU) in modern processors
SRRIP: the use of multiple bits per block to track priorities at a finer granularity.
Incoming blocks are being given more time to determine their re-use potential.
DRRIP (BRRIP+SRRIP) using Set-Dueling
DRRIP outperform LRU 10% with 2X less hardware overhead.
## Behavior of LRU, NRU and SRRIP

<table>
<thead>
<tr>
<th>Next Ref</th>
<th>RRIP head</th>
<th>RRIP tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$1 \rightarrow 1 \rightarrow 1 \rightarrow 1$</td>
<td>miss</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$a_1 \rightarrow 1 \rightarrow 1 \rightarrow 1$</td>
<td>miss</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$a_2 \rightarrow a_1 \rightarrow 1 \rightarrow 1$</td>
<td>hit</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$a_1 \rightarrow a_2 \rightarrow 1 \rightarrow 1$</td>
<td>miss</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$b_1 \rightarrow a_1 \rightarrow a_2 \rightarrow 1$</td>
<td>miss</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$b_2 \rightarrow b_1 \rightarrow a_1 \rightarrow a_2$</td>
<td>miss</td>
</tr>
<tr>
<td>$b_4$</td>
<td>$b_3 \rightarrow b_2 \rightarrow b_1 \rightarrow a_1$</td>
<td>miss</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_4 \rightarrow b_3 \rightarrow b_2 \rightarrow b_1$</td>
<td>miss</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$a_1 \rightarrow b_4 \rightarrow b_3 \rightarrow b_2$</td>
<td>miss</td>
</tr>
</tbody>
</table>

(a) LRU  
(b) Not Recently Used (NRU)  
(c) 2-bit SRRIP with Hit Promotion

---

Optimal Belady and Belady+Bypassing

Belady Replacement
- Optimal longest-forward-distance replacement

Belady-B:
- Belady extended with bypass which drops an incoming block if its next forward use distance is larger than all the blocks in the target LLC set

Base-B: NRU-bypassing

Bypassing is good.

**SHiP. C. Wu, A. Jaleel et al. MICRO’11**

**Signature-based Hit Predictor**
- Memory Region
- Program Counter
- Instruction History

**Insertion Policy**
- If \((\text{SHCT}[\text{sig}] == 0)\) 3;
- Always 2 in SRRIP.

```
if hit then
    cache_line.outcome = true;
    Increment SHCT[signature_m];
else
    if evicted_cache_line.outcome != true
        Decrement SHCT[signature_m];
    cache_line.outcome = false;
    cache_line.signature_m = signature;
    if SHCT[signature] == 0
        Predict distant re-reference;
    else
        Predict intermediate re-reference;
end if
```
PDP: Protecting Distance. MICRO’12

Policy: Prevents replacing a cache line until a certain number of accesses to its cache set, called Protecting Distance (PD)

Bypassing when protecting

Reuse Distance (RD)
- The number of accesses to a cache set between two accesses to the same cache line
- Not to evict lines too early
- Not to keep lines in the cache for too long
- Reuse Distance distribution
Dead Blocks

Access Count Predictor, IEEE TOC 2008 (131)
- Event counter very block in L2
- When counter exceeds a threshold, the block dead.
- Bypassing

Cache Burst, MICRO 2008 (108)
Virtual Victim Cache, PACT 2010
Sampling Dead Block Prediction, MICRO 2010
Fig. 2: Refract dead block predictor (a), and new dead block predictor with sampler tag array (b). The sampler and dead block predictor table are updated for 1.6% of the accesses to the LLC, while the refract predictor is updated on every access.

Fig. 3: Block diagrams for dead block predictors

index = blk.signature

confidence

dead if confidence >= threshold

index1 = hash1(blk.signature)
index2 = hash2(blk.signature)
index3 = hash3(blk.signature)

confidence1
confidence2
confidence3

dead if confidence1+confidence2+confidence3 >= threshold
### OPT-Violation Assertion, PACT'12

<table>
<thead>
<tr>
<th>Action on IB-VB pair</th>
<th>Reuse distance relationship</th>
<th>optimal bypass behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion 1</td>
<td>Current incoming block hits IB.</td>
<td>IB’s reuse distance &lt; VB’s reuse distance</td>
</tr>
<tr>
<td>Assertion 2</td>
<td>Current incoming block hits VB.</td>
<td>IB’s reuse distance &gt; VB’s reuse distance</td>
</tr>
<tr>
<td>Assertion 3</td>
<td>Current victim block hits IB.</td>
<td>IB’s reuse distance &gt; cache size and VB’s reuse distance &gt; cache size</td>
</tr>
</tbody>
</table>

**Figure 5: The structure of OBM.**

**Figure 6: The algorithm of OBM.**
State of the art replacement policies only achieve average within 5% improvement of NRU.

Decoupled tag/data SLLC which only store the data of lines that reused.

With Tag Size of 4MB $\rightarrow$ Data Size of 1MB $\rightarrow$ perform as well as 8MB cache with 16.7% storage.

Observation: if with the OPT solution a load instruction has historically brought in lines that produce cache hits, then in the future, the same load instruction is likely to bring in lines that will also produce a cache hit.

Usage interval: from demand to next reference

Liveness intervals: the time period during which that line resides in the cache under the OPT policy.

One would be a cache miss if at any point in its usage interval the number of overlapping liveness intervals matches the cache’s capacity.
Hawkeye ISCA’16

**OPTgen**
*Computes optimal solution for the past*

**Hawkeye Predictor**
*Learns from OPT to make future predictions*

**Access Sequence**

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Timeline view of the Access Stream**

**Optimal Solution (4 hits)**
[Cache hits marked as solid lines]

**OPTgen Solution (4 hits)**
[State of the Occupancy Vector over time]
Summary

MIP: MRU insertion policy (traditional approach)

DIP: Selects the best

LIP: LRU insertion policy

BIP: Bimodal insertion policy
Few insertions at head, most at tail

RRIP: Probabilistic insertion near tail

TADIP: Selects the best for each thread

PIPP: Inserts each thread at different positions + probabilistic promotion

AGGRESSOR-VT: Victimizes the aggressor thread with a high probability

UCP: Partitions ways across threads based on marginal utility

Highest priority ← Priority stack of blocks in a set → Lowest priority