Computer Architecture
Lecture 19a:
Multi-Core Cache Management II

Prof. Onur Mutlu
ETH Zürich
Fall 2018
28 November 2018
Efficient Cache Utilization: Examples


Controlled Shared Caching
Hardware-Based Cache Partitioning
Software-Based Shared Cache Partitioning
Private/Shared Caching
Private/Shared Caching

- **Goal:** Achieve the benefits of private caches (low latency, performance isolation) while sharing cache capacity across cores

- **Example:** Adaptive spill/receive caching

- **Idea:** Start with a private cache design (for performance isolation), but dynamically steal space from other cores that do not need all their private caches
  - Some caches can spill their data to other cores’ caches dynamically

Private caches avoid the need for shared interconnect ++ fast latency, tiled design, performance isolation

Problem: When one core needs more cache and other core has spare cache, private-cache based systems cannot share capacity
Cache Line Spilling – Cooperative Caching

Spill evicted line from one cache to neighbor cache
- Co-operative caching (CC) [Chang+ ISCA’06]

Problem with CC:
1. Performance depends on the parameter (spill probability)
2. All caches spill as well as receive → Limited improvement

Goal: Robust High-Performance Capacity Sharing with Negligible Overhead

Spill-Receive Architecture

Each Cache is either a Spiller or Receiver but not both
- Lines from spiller cache are spilled to one of the receivers
- Evicted lines from receiver cache are discarded

What is the best N-bit binary string that maximizes the performance of Spill Receive Architecture ➔ Dynamic Spill Receive (DSR)

Dynamic Spill-Receive via “Set Dueling”

Divide the cache in three:
- Spiller sets
- Receiver sets
- Follower sets (winner of spiller, receiver)

n-bit PSEL counter
misses to spiller-sets: PSEL--
misses to receiver-set: PSEL++

MSB of PSEL decides policy for Follower sets:
- MSB = 0, Use spill
- MSB = 1, Use receive

monitor ➔ choose ➔ apply (using a single counter)
Dynamic Spill-Receive Architecture

Each cache learns whether it should act as a spiller or receiver.

Decides policy for all sets of Cache A (except X and Y)
Experimental Setup

- **Baseline Study:**
  - 4-core CMP with in-order cores
  - Private Cache Hierarchy: 16KB L1, 1MB L2
  - 10 cycle latency for local hits, 40 cycles for remote hits

- **Benchmarks:**
  - 6 benchmarks that have extra cache: “Givers” (G)
  - 6 benchmarks that benefit from more cache: “Takers” (T)
  - All 4-thread combinations of 12 benchmarks: 495 total

Five types of workloads: G4T0 G3T1 G2T2 G1T3 G0T4
Results for Weighted Speedup

On average, DSR improves weighted speedup by 13%
Distributed Caches

**FIGURE 1. Typical tiled architecture.** Tiles are interconnected into a 2-D folded torus. Each tile contains a core, L1 instruction and data caches, a shared-L2 cache slice, and a router/switch.
Caching for Parallel Applications

Data placement determines performance
Goal: place data on chip close to where they are used
Handling Shared Data in Private Caches

- Shared data and locks ping-pong between processors if caches are private
  -- Increases latency to fetch shared data/locks
  -- Reduces cache efficiency (many invalid blocks)
  -- Scalability problem: maintaining coherence across a large number of private caches is costly

How to do better?
- Idea: Store shared data and locks only in one special core’s cache. Divert all critical section execution to that core/cache.
  - Essentially, a specialized core for processing critical sections
Non-Uniform Cache Access

- Problem: Large caches take a long time to access
- Wire delay
  - Closeby blocks can be accessed faster, but furthest blocks determine the worst-case access time

- Idea: **Variable latency access time in a single cache**
- **Partition cache into pieces**
  - Each piece has different latency
  - Which piece does an address map to?
    - Static: based on bits in address
    - Dynamic: any address can map to any piece
      - How to locate an address?
      - Replacement and placement policies?

Multi-Core Cache Efficiency: Bandwidth Filters

- Caches act as a filter that reduce memory bandwidth requirement
  - Cache hit: No need to access memory
  - This is in addition to the latency reduction benefit of caching
  - GPUs use caches to reduce memory BW requirements

- Efficient utilization of cache space becomes more important with multi-core
  - Memory bandwidth is more valuable
    - Pin count not increasing as fast as # of transistors
      - 10%/year vs. 2x every 2 years
  - More cores put more pressure on the memory bandwidth

- How to make the bandwidth filtering effect of caches better?
Efficient Cache Utilization
Efficient Cache Utilization: Examples


Cache Compression
Motivation for Cache Compression

Significant redundancy in data:

| 0x00000000 | 0x0000000B | 0x00000003 | 0x00000004 | ... |

How can we exploit this redundancy?

– **Cache compression** helps

– Provides effect of a larger cache without making it physically larger
Background on Cache Compression

• Key requirements:
  – **Fast** (low decompression latency)
  – **Simple** (avoid complex hardware changes)
  – **Effective** (good compression ratio)
## Summary of Major Works

<table>
<thead>
<tr>
<th>Compression Mechanisms</th>
<th>Decompression Latency</th>
<th>Complexity</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
## Summary of Major Works

<table>
<thead>
<tr>
<th>Compression Mechanisms</th>
<th>Decompression Latency</th>
<th>Complexity</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Frequent Value</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Summary of Major Works

<table>
<thead>
<tr>
<th>Compression Mechanisms</th>
<th>Decompression Latency</th>
<th>Complexity</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Frequent Value</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Frequent Pattern</td>
<td>✗</td>
<td>✗ / ✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
# Summary of Major Works

<table>
<thead>
<tr>
<th>Compression Mechanisms</th>
<th>Decompression Latency</th>
<th>Complexity</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Frequent Value</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Frequent Pattern</td>
<td>✗</td>
<td>✗ / ✓</td>
<td>✓</td>
</tr>
<tr>
<td>BΔI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Base-Delta-Immediate Cache Compression

Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Philip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry,
"Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches"
Executive Summary

• Off-chip memory latency is high
  – Large caches can help, but at significant cost
• Compressing data in cache enables larger cache at low cost
• **Problem**: Decompression is on the execution critical path
• **Goal**: Design a new compression scheme that has
  1. low decompression latency, 2. low cost, 3. high compression ratio
• **Observation**: Many cache lines have low dynamic range data
• **Key Idea**: Encode cachelines as a base + multiple differences
• **Solution**: Base-Delta-Immediate compression with low decompression latency and high compression ratio
  – Outperforms three state-of-the-art compression mechanisms
Key Data Patterns in Real Applications

**Zero Values:** initialization, sparse matrices, NULL pointers

| 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | ... |

**Repeated Values:** common initial values, adjacent pixels

| 0x000000FF | 0x000000FF | 0x000000FF | 0x000000FF | ... |

**Narrow Values:** small values stored in a big data type

| 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | ... |

**Other Patterns:** pointers to the same memory region

| 0xC04039C0 | 0xC04039C8 | 0xC04039D0 | 0xC04039D8 | ... |
### How Common Are These Patterns?

SPEC2006, databases, web workloads, 2MB L2 cache

“Other Patterns” include Narrow Values

| SPEC2006, databases, web workloads, 2MB L2 cache | 43% of the cache lines belong to key patterns |

#### Average Cache Coverage (%)

- **Zero**: Zero
- **Repeated Values**: Repeated Values
- **Other Patterns**: Other Patterns

#### Graphical Representation


- **Average**: Average
Key Data Patterns in Real Applications

Low Dynamic Range:

Differences between values are significantly smaller than the values themselves
Key Idea: Base+Delta (B+Δ) Encoding

32-byte Uncompressed Cache Line

0xC04039C0 0xC04039C8 0xC04039D0 ... 0xC04039F8

12-byte Compressed Cache Line

0x00 0x08 0x10 ... 0x38

4 bytes

Base

✓ Fast Decompression: vector addition

✓ Simple Hardware: arithmetic and comparison

✓ Effective: good compression ratio
### Can We Do Better?

- **Uncompressible cache line (with a single base):**
  
  | 0x00000000 | 0x09A40178 | 0x0000000B | 0x09A4A838 | ... |

- **Key idea:**
  Use more bases, e.g., two instead of one
  - **Pro:**
    - More cache lines can be compressed
  - **Cons:**
    - Unclear how to find these bases efficiently
    - Higher overhead (due to additional bases)
B+Δ with Multiple Arbitrary Bases

![Bar chart showing compression ratios for GeoMean with different bases. The chart indicates that 2 bases is the best option based on evaluations.](chart.png)
How to Find Two Bases Efficiently?

1. **First base** - first element in the cache line
   - Base+Delta part

2. **Second base** - implicit base of 0
   - Immediate part

Advantages over 2 arbitrary bases:
- Better compression ratio
- Simpler compression logic

**Base-Delta-Immediate (BΔI) Compression**
B+Δ (with two arbitrary bases) vs. BΔI

Average compression ratio is close, but BΔI is simpler
BΔI Cache Compression Implementation

• Decompressor Design
  – Low latency

• Compressor Design
  – Low cost and complexity

• BΔI Cache Organization
  – Modest complexity
**BΔI Decompressor Design**

Compressed Cache Line

Uncompressed Cache Line

Vector addition
BΔI Compressor Design

Compression Selection Logic (based on compr. size)

Compressed Cache Line
BΔI Compression Unit: 8-byte $B_0$ 1-byte $\Delta$

32-byte Uncompressed Cache Line

8 bytes

$V_0 \quad V_0 \quad V_1 \quad V_2 \quad V_3$

$B_0 = V_0 \quad B_0 \quad B_0 \quad B_0 \quad B_0$

$\Delta_0 \quad \Delta_1 \quad \Delta_2 \quad \Delta_3$

Within 1-byte range?  Within 1-byte range?  Within 1-byte range?  Within 1-byte range?

Is every element within 1-byte range?

Yes  No
**BΔI Cache Organization**

**Data Storage:** Conventional 2-way cache with 32-byte cache lines

**Tag Storage:**

Set0  
...  ...  ...
Set1  
Tag0  Tag1  ...
...  ...
Way0  Way1

**Data Storage:**

Set0  
...  ...
Set1  
Data0  Data1  ...
...  ...
Way0  Way1

**BΔI: 4-way cache with 8-byte segmented data**

**Tag Storage:**

Set0  
...  ...
Set1  
Tag0  Tag1  Tag2  Tag3  ...
...  ...
Way0  Way1  Way2  Way3  

**Data Storage:**

Set0  
...  ...
Set1  
S0  S1  S2  S3  S4  S5  S6  S7  ...
...  ...
Way0  Way1  Way2  Way3  

Twice as many tags, C - Comor. encoding bits

3% multiple read for 2 segment cache
Qualitative Comparison with Prior Work

- **Zero-based designs**
  - ZCA [Dusser+, ICS’09]: zero-content augmented cache
  - ZVC [Islam+, PACT’09]: zero-value cancelling
    - Limited applicability (only zero values)
- **FVC** [Yang+, MICRO’00]: frequent value compression
  - High decompression latency and complexity
- **Pattern-based compression designs**
  - FPC [Alameldeen+, ISCA’04]: frequent pattern compression
    - High decompression latency (5 cycles) and complexity
  - C-pack [Chen+, T-VLSI Systems’10]: practical implementation of FPC-like algorithm
    - High decompression latency (8 cycles)
BΔI achieves the highest compression ratio
**Single-Core: IPC and MPKI**

**Normalized IPC**

- Baseline (no compr.)
- BΔl

**L2 cache size**

- 512kB
- 1MB
- 2MB
- 4MB
- 8MB
- 16MB

- 8.1%
- 5.2%
- 5.1%
- 4.9%
- 5.6%
- 3.6%

**Normalized MPKI**

- Baseline (no compr.)
- BΔl

**L2 cache size**

- 512kB
- 2MB
- 8MB

- 16%
- 24%
- 21%
- 13%
- 19%
- 14%

**BΔl** achieves the performance of a 2X-size cache

Performance improves due to the decrease in MPKI
Multi-Core Workloads

- Application classification based on **Compressibility**: effective cache size increase
  (Low Compr. ($LC$) < 1.40, High Compr. ($HC$) >= 1.40)

- **Sensitivity**: performance gain with more cache
  (Low Sens. ($LS$) < 1.10, High Sens. ($HS$) >= 1.10; 512kB -> 2MB)

- Three classes of applications:
  - LCLS, HCLS, HCHS, **no LCHS** applications

- For 2-core - **random** mixes of each possible class pairs
  (20 each, 120 total workloads)
Multi-Core: Weighted Speedup

If at least one application is sensitive then the performance improves (9.5%).
Other Results in Paper

- IPC comparison against *upper* bounds
  - $B\Delta I$ almost achieves performance of the 2X-size cache
- Sensitivity study of having *more* than 2X tags
  - Up to 1.98 average compression ratio
- Effect on *bandwidth* consumption
  - 2.31X decrease on average
- Detailed quantitative comparison with prior work
- **Cost analysis** of the proposed changes
  - 2.3% L2 cache area increase
Conclusion

• A new **Base-Delta-Immediate** compression mechanism

• **Key insight**: many cache lines can be efficiently represented using **base + delta encoding**

• **Key properties:**
  - **Low** latency decompression
  - **Simple** hardware implementation
  - **High compression ratio** with high coverage

• **Improves** *cache hit ratio* and *performance* of both single-core and multi-core workloads
  - Outperforms state-of-the-art cache compression techniques: FVC and FPC
Readings on Memory Compression (I)

- Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Philip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry,

"Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches"


Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches

Gennady Pekhimenko†
gpekhime@cs.cmu.edu

Vivek Seshadri†
vsejadhr@cs.cmu.edu

Onur Mutlu†
onur@cmu.edu

Michael A. Kozuch*
michael.a.kozuch@intel.com

Phillip B. Gibbons*
phillip.b.gibbons@intel.com

Todd C. Mowry†
tcm@cs.cmu.edu

†Carnegie Mellon University  *Intel Labs Pittsburgh
Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework"
Readings on Memory Compression (III)

- Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip P. Gibbons, Michael A. Kozuch, and Todd C. Mowry, "Exploiting Compressed Block Size as an Indicator of Future Reuse"

Readings on Memory Compression (IV)


A Case for Toggle-Aware Compression for GPU Systems

Gennady Pekhimenko†, Evgeny Bolotin*, Nandita Vijaykumar†, Onur Mutlu†, Todd C. Mowry†, Stephen W. Keckler*#

†Carnegie Mellon University  *NVIDIA  #University of Texas at Austin
Readings on Memory Compression (VI)

- Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu,

"A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps"
[Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)]
Efficient Cache Utilization: Examples

Is inserting a fetched/prefetched block into the cache (hierarchy) always a good idea?
- No allocate on write: does not allocate a block on write miss
- How about reads?

Allocating on a read miss
- Evicts another potentially useful cache block
- Incoming block potentially more useful

Ideally:
- we would like to place those blocks whose caching would be most useful in the future
- we certainly do not want to cache never-to-be-used blocks
Revisiting Cache Placement (Insertion)

- **Ideas:**
  - **Hardware** predicts blocks that are not going to be used
  - **Software** (programmer/compiler) marks instructions that touch data that is not going to be reused
    - How does software determine this?

- **Streaming versus non-streaming accesses**
  - If a program is streaming through data, reuse likely occurs only for a limited period of time
  - If such instructions are marked by the software, the hardware can store them temporarily in a smaller buffer (L0 cache) instead of the cache
Reuse at L2 Cache Level

DoA Blocks: Blocks unused between insertion and eviction

For the 1MB 16-way L2, 60% of lines are DoA

⇒ Ineffective use of cache space
Why Dead on Arrival Blocks?

- Streaming data ➔ Never reused. L2 caches don’t help.
- Working set of application greater than cache size

Solution: if working set > cache size, retain some working set
Cache Insertion Policies: MRU vs. LRU

Reference to ‘i’ with traditional LRU policy:

Reference to ‘i’ with LIP (LRU Insertion Policy):

Choose victim. Do NOT promote to MRU

Lines do not enter non-LRU positions unless reused
Other Insertion Policies: Bimodal Insertion

LIP does not age older lines

Infrequently insert lines in MRU position

Let $\varepsilon = \text{Bimodal throttle parameter}$

```plaintext
if ( rand() < \varepsilon )
    Insert at MRU position;
else
    Insert at LRU position;
```

For small $\varepsilon$, BIP retains thrashing protection of LIP while responding to changes in working set
Analysis with Circular Reference Model

Reference stream has T blocks and repeats N times. Cache has K blocks (K<T and N>>T)

Cache hit rates of two consecutive reference streams:

<table>
<thead>
<tr>
<th>Policy</th>
<th>((a_1 a_2 a_3 \ldots a_T)^N)</th>
<th>((b_1 b_2 b_3 \ldots b_T)^N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OPT</td>
<td>((K-1)/T)</td>
<td>((K-1)/T)</td>
</tr>
<tr>
<td>LIP</td>
<td>((K-1)/T)</td>
<td>0</td>
</tr>
<tr>
<td>BIP (small (\varepsilon))</td>
<td>(\approx (K-1)/T)</td>
<td>(\approx (K-1)/T)</td>
</tr>
</tbody>
</table>

For small \(\varepsilon\), BIP retains thrashing protection of LIP while adapting to changes in working set.
### Analysis with Circular Reference Model

#### Table 3: Hit Rate for LRU, OPT, LIP, and BIP

<table>
<thead>
<tr>
<th></th>
<th>((a_1 \cdots a_T)^N)</th>
<th>((b_1 \cdots b_T)^N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OPT</td>
<td>((K - 1)/T)</td>
<td>((K - 1)/T)</td>
</tr>
<tr>
<td>LIP</td>
<td>((K - 1)/T)</td>
<td>0</td>
</tr>
<tr>
<td>BIP</td>
<td>((K - 1 - \epsilon \cdot [T - K])/T)</td>
<td>((K - 1 - \epsilon \cdot [T - K])/T)</td>
</tr>
<tr>
<td></td>
<td>(\approx (K - 1)/T)</td>
<td>(\approx (K - 1)/T)</td>
</tr>
</tbody>
</table>
LIP and BIP Performance vs. LRU

Changes to insertion policy increases misses for LRU-friendly workloads.
Dynamic Insertion Policy (DIP)


Two types of workloads: LRU-friendly or BIP-friendly

DIP can be implemented by:

1. **Monitor** both policies (LRU and BIP)
2. **Choose** the best-performing policy
3. **Apply** the best policy to the cache

Need a cost-effective implementation ➔ Set Sampling
Dynamic Insertion Policy Miss Rate

(BIP) DIP (32 dedicated sets)

(%) Reduction in L2 MPKI

art mcf twolf vpr facerec ammp galgel equake bzip2 parser sixtrack apsi lucas mgrid swim health amean
Efficient Cache Utilization: Examples


Cache Utilization is Important

Increasing contention

Effective cache utilization is important
Reuse Behavior of Cache Blocks

Different blocks have different reuse behavior

Access Sequence:

- High-reuse block
- Low-reuse block

Ideal Cache
Cache Pollution

**Problem:** Low-reuse blocks evict high-reuse blocks

**Idea:** Predict reuse behavior of missed blocks. Insert low-reuse blocks at LRU position.
Cache Thrashing

**Problem:** High-reuse blocks evict each other

**Idea:** Insert at MRU position with a very low probability (*Bimodal insertion policy*)

A fraction of working set stays in cache

Handling Pollution and Thrashing

Need to address both pollution and thrashing concurrently

**Cache Pollution**
Need to distinguish high-reuse blocks from low-reuse blocks

**Cache Thrashing**
Need to control the number of blocks inserted with high priority into the cache
Reuse Prediction

Miss

Missed-block

High reuse

Low reuse

Keep track of the reuse behavior of every cache block in the system

Impractical
1. High storage overhead
2. Look-up latency
Approaches to Reuse Prediction

Use program counter or memory region information.

1. Group Blocks
   - PC 1: AB
   - PC 2: ST

2. Learn group behavior
   - PC 1: AB
   - PC 2: ST

3. Predict reuse
   - PC 1: C
   - PC 2: U

1. Same group $\not\rightarrow$ same reuse behavior
2. No control over number of high-reuse blocks
Per-block Reuse Prediction

Use recency of eviction to predict reuse

Time of eviction

Accessed soon after eviction

Accessed long time after eviction
Evicted-Address Filter (EAF)

Evicted-block address

EAF

Addresses of recently evicted blocks

Cache

MRU

LRU

In EAF?

Yes

No

High Reuse

Low Reuse

Missed-block address

Miss
Naïve Implementation: Full Address Tags

1. Large storage overhead
2. Associative lookups – High energy

Recently evicted address
Need not be 100% accurate
Low-Cost Implementation: Bloom Filter

Implement EAF using a **Bloom Filter**
Low storage overhead + energy

Need not be 100% accurate
Bloom Filters (From Lecture 1)

Space/Time Trade-offs in Hash Coding with Allowable Errors

Burton H. Bloom

In this paper trade-offs among certain computational factors in hash coding are analyzed. The paradigm problem considered is that of testing a series of messages one-by-one for membership in a given set of messages. Two new hash-coding methods are examined and compared with a particular conventional hash-coding method. The computational factors considered are the size of the hash area (space), the time required to identify a message as a nonmember of the given set (reject time), and an allowable error frequency.

Bloom Filter

Compact representation of a set
1. Bit vector
2. Set of hash functions

May remove multiple addresses

Inserted Elements: X  Y
EAF using a Bloom Filter

- **Insert**: Evicted-block address
- **Test**: Missed-block address
- **Remove**: If present

**EAF**

1. **Remove** not supported
2. **Clear** when full

Bloom-filter EAF: 4x reduction in storage overhead, 1.47% compared to cache size
EAF-Cache: Final Design

1. **Cache eviction**
   - Insert address into filter
   - Increment counter

2. **Cache miss**
   - Test if address is present in filter
   - Yes, insert at MRU. No, insert with BIP

3. **Counter reaches max**
   - Clear filter and counter
EAF: Advantages

1. Simple to implement
2. Easy to design and verify
3. Works with other techniques (replacement policy)
EAF Performance – Summary

Performance Improvement over LRU

- TA-DIP
- TA-DRRIP
- RTB
- MCT
- SHIP
- EAF
- D-EAF

1-Core

2-Core

4-Core
Comparison with Prior Works

Addressing Cache Pollution
Run-time Bypassing (RTB) – Johnson+ ISCA’97
- Memory region based reuse prediction

Single-usage Block Prediction (SU) – Piquet+ ACSAC’07
Signature-based Hit Prediction (SHIP) – Wu+ MICRO’11
- Program counter based reuse prediction

Miss Classification Table (MCT) – Collins+ MICRO’99
- One most recently evicted block

- No control on number of blocks inserted with high priority $\Rightarrow$ Thrashing
Comparison with Prior Works

Addressing Cache Thrashing

TA-DIP – Qureshi+ ISCA’07, Jaleel+ PACT’08
TA-DRRIP – Jaleel+ ISCA’10

- Use set dueling to determine thrashing applications
- No mechanism to filter low-reuse blocks ⇒ Pollution
4-Core: Performance

Weighted Speedup Improvement over LRU

LRU
EAF
SHIP
D-EAF

Workload Number (135 workloads)
Effect of Cache Size

Weighted Speedup Improvement over LRU

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>2-Core</th>
<th>4-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>1MB</td>
<td><img src="image1" alt="Graph for 1MB 2-Core" /></td>
<td><img src="image2" alt="Graph for 1MB 4-Core" /></td>
</tr>
<tr>
<td>2MB</td>
<td><img src="image3" alt="Graph for 2MB 2-Core" /></td>
<td><img src="image4" alt="Graph for 2MB 4-Core" /></td>
</tr>
<tr>
<td>4MB</td>
<td><img src="image5" alt="Graph for 4MB 2-Core" /></td>
<td><img src="image6" alt="Graph for 4MB 4-Core" /></td>
</tr>
<tr>
<td>8MB</td>
<td><img src="image7" alt="Graph for 8MB 2-Core" /></td>
<td><img src="image8" alt="Graph for 8MB 4-Core" /></td>
</tr>
<tr>
<td>16MB</td>
<td><img src="image9" alt="Graph for 16MB 2-Core" /></td>
<td><img src="image10" alt="Graph for 16MB 4-Core" /></td>
</tr>
</tbody>
</table>

Legend:
- SHIP
- EAF
- D-EAF
Effect of EAF Size

The graph shows the weighted speedup improvement over LRU for different EAF size ratios. The x-axis represents the number of addresses in EAF divided by the number of blocks in the cache, ranging from 0 to 1.6. The y-axis represents the percentage improvement in weighted speedup.

- The red line represents a 1-core system, showing a steady increase in improvement with an increase in EAF size.
- The yellow line represents a 2-core system, which also shows an increase in improvement but at a steeper rate compared to the 1-core system.
- The black line represents a 4-core system, which shows the highest improvement rate among the three.

Overall, the graph illustrates how the weighted speedup improvement over LRU increases with the size of the EAF, especially for systems with more cores.
Other Results in Paper

- EAF orthogonal to replacement policies
  - LRU, RRIP – Jaleel+ ISCA’10
- Performance improvement of EAF increases with increasing memory latency
- EAF performs well on four different metrics
  - Performance and fairness
- Alternative EAF-based designs perform comparably
  - Segmented EAF
  - Decoupled-clear EAF
More on Evicted Address Filter Cache

- Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing"

The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing

Vivek Seshadri† vseshadr@cs.cmu.edu  Onur Mutlu† onur@cmu.edu  Michael A Kozuch* michael.a.kozuch@intel.com  Todd C Mowry† tcm@cs.cmu.edu

†Carnegie Mellon University  *Intel Labs Pittsburgh
Predictable Performance Again: Strong Memory Service Guarantees
Remember MISE?

- Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu,

"MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems"

Extending Slowdown Estimation to Caches

- How do we extend the MISE model to include shared cache interference?

- Answer: Application Slowdown Model

- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,
  "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"
  Proceedings of the 48th International Symposium on Microarchitecture (MICRO), Waikiki, Hawaii, USA, December 2015.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
  [Source Code]
Application Slowdown Model

Quantifying and Controlling Impact of Interference at Shared Caches and Main Memory

Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, Onur Mutlu
Shared Cache and Memory Contention

Slowdown = \frac{\text{Request Service Rate Alone}}{\text{Request Service Rate Shared}}

MISE [HPCA'13]
Cache Capacity Contention

Applications evict each others’ blocks from the shared cache
Estimating Cache and Memory Slowdowns
Service Rates vs. Access Rates

Request service and access rates are tightly coupled.
The Application Slowdown Model

Slowdown = \frac{\text{Cache Access Rate} \text{ Alone}}{\text{Cache Access Rate} \text{ Shared}}
Real System Studies: Cache Access Rate vs. Slowdown

- **Cache Access Rate Ratio**
- **Slowdown**

Graph showing the relationship between cache access rate ratio and slowdown for different applications:
- *astar*
- *lbm*
- *bzip2*
Challenge

How to estimate alone cache access rate?
Auxiliary Tag Store

Auxiliary tag store tracks such *contention misses*
Accounting for Contention Misses

• Revisiting alone memory request service rate

Alone Request Service Rate of an Application =
\[
\frac{\text{# Requests During High Priority Epochs}}{\text{# High Priority Cycles}}
\]

*Cycles serving contention misses should not count as high priority cycles*
Alone Cache Access Rate Estimation

Cache Access Rate Alone of an Application =

\[
\frac{\# \text{ Requests During High Priority Epochs}}{\# \text{ High Priority Cycles} - \# \text{Cache Contention Cycles}}
\]

**Cache Contention Cycles:** Cycles spent serving contention misses

Cache Contention Cycles = \# Contention Misses \times Average Memory Service Time

*From auxiliary tag store when given high priority*

*Measured when given high priority*
Application Slowdown Model (ASM)

\[ \text{Slowdown} = \frac{\text{Cache Access Rate Alone}}{\text{Cache Access Rate Shared}} \]
Previous Work on Slowdown Estimation

• Previous work on slowdown estimation
  – **STFM** (Stall Time Fair Memory) Scheduling [Mutlu et al., MICRO ’07]
  – **FST** (Fairness via Source Throttling) [Ebrahimi et al., ASPLOS ’10]
  – **Per-thread Cycle Accounting** [Du Bois et al., HiPEAC ’13]

• Basic Idea:

\[
\text{Slowdown} = \frac{\text{Execution Time Alone}}{\text{Execution Time Shared}}
\]

Count interference experienced by each request \(\rightarrow\) Difficult
ASM’s estimates are much more coarse grained \(\rightarrow\) Easier
Model Accuracy Results

Average error of ASM’s slowdown estimates: 10%
Leveraging ASM’s Slowdown Estimates

- **Slowdown-aware resource allocation for high performance and fairness**
- **Slowdown-aware resource allocation to bound application slowdowns**
- **VM migration and admission control schemes** [VEE ’15]
- **Fair billing schemes in a commodity cloud**
Goal: Partition the shared cache among applications to mitigate contention
Cache Capacity Partitioning

Previous partitioning schemes optimize for miss count

Problem: Not aware of performance and slowdowns
ASM-Cache: Slowdown-aware Cache Way Partitioning

- **Key Requirement:** Slowdown estimates for all possible way partitions

- Extend ASM to estimate slowdown for all possible cache way allocations

- **Key Idea:** Allocate each way to the application whose slowdown reduces the most
Memory Bandwidth Partitioning

Goal: Partition the main memory bandwidth among applications to mitigate contention
ASM-Mem: Slowdown-aware Memory Bandwidth Partitioning

- **Key Idea:** Allocate high priority proportional to an application’s slowdown

\[
\text{High Priority Fraction}_i = \frac{\text{Slowdown}_i}{\sum_j \text{Slowdown}_j}
\]

- Application i’s requests given highest priority at the memory controller for its fraction
Coordinated Resource Allocation Schemes

1. Employ ASM-Cache to partition cache capacity
2. Drive ASM-Mem with slowdowns from ASM-Cache
Significant fairness benefits across different channel counts
Summary

• Problem: Uncontrolled memory interference cause high and unpredictable application slowdowns
• Goal: Quantify and control slowdowns
• Key Contribution:
  – ASM: An accurate slowdown estimation model
  – Average error of ASM: 10%
• Key Ideas:
  – Shared cache access rate is a proxy for performance
  – Cache Access Rate \(A_{\text{alone}}\) can be estimated by minimizing memory interference and quantifying cache interference
• Applications of Our Model
  – Slowdown-aware cache and memory management to achieve high performance, fairness and performance guarantees
• Source Code Released in January 2016
More on Application Slowdown Model

- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,

"The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"

Proceedings of the 48th International Symposium on Microarchitecture (MICRO), Waikiki, Hawaii, USA, December 2015.

[Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
[Source Code]