Shared Cache

Scott Morgan

September 8, 2015
About Me:
- First Year MCS Student
- UNC Chapel Hill ’15
- Statistics/Operations Research and Mathematics

Disclaimer:
- First course having anything to do with hardware
- Couldn’t have told you what a cache was before this class
Papers Covered

Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors

- M. Zhang and K. Asanovic.
- Proceedings 32nd International Symposium on Computer Architecture, Madison, WI
- June 2005

Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors

- Enric Herrero, Jose Gonzalez, Ramon Canal
- International Symposium on Computer Architecture, Saint-Malo, France
- June 2010
Outline of Presentation

- Motivation
- Primer on Caching
- Victim Replication
- Elastic Cooperative Caching
- Questions
Motivation

Problem: Chip multiprocessors had “dance-hall” architecture (ex. Niagara)
  - Doesn’t scale well to large core counts
Solution: Use tiled architecture
  - How do you manage the memory hierarchy?
Primer on Caching
Cache Coherence: the consistency of shared data stored in multiple local caches

source: wikipedia
**Definition**

**Cache Coherence:** Unless care is taken, a coherence problem can arise if multiple actors (e.g., multiple cores) have access to multiple copies of a datum (e.g., in multiple caches) and at least one access is a write.

source: A Primer on Memory Consistency and Cache Coherence
Coherence Protocol

MESI Protocol

- **Modified**: only in one cache, but different from value in memory. Must perform a write-back to main memory, after which the state will change to **exclusive**
- **Exclusive**: only in one cache and equal to value in memory
- **Shared**: may be in another cache, but it is equal to value in memory
- **Invalid**: the cache line is invalid
**Impact of Operations**

- **Read**: can read from Modified, Shared, or Exclusive.
  - If Modified, change to Shared and write back to main memory
  - If Exclusive, change to Shared

- **Write**: can only write when Modified or Exclusive. If Shared, the all other copied must be invalidated, after which it will become Exclusive.

- **Discard**: Shared or Exclusive lines can be discarded. Modified lines must be written back first.

**source**: wikipedia
Introduction

- CMPs are useful for applications with thread-level parallelism
- Tiled CMPs scale well
- But, the more tiles, the higher the latencies if caches are shared
- In future applications, bigger caches will be needed

How should the L2 cache be managed?
Two basic ways to manage the L2 cache:

- **L2P**: Each processor has a slice of the L2 cache that is private
  - Low hit latency, but only if the working set fits into the local cache slice
  - Each processor keeps a local copy of the data, so many unnecessary copies
  - Large working sets introduce high latencies

- **L2S**: Each processor has a slice of the L2 cache, each of which is part of a distributed L2 cache that they all share
  - Low off-chip miss rate
  - High latencies introduced by having to retrieve data from across the chip
Victim Replication (L2VR): “a hybrid cache management policy which combines the advantages of both private and shared L2 scheme”.

- Based on L2S management scheme
- Multiple copies of a data line can exist in L2 cache
- Each data line has a **home tile** and each copy not there is a **replica**
- “In effect, replicas form dynamically-created private L2 caches to reduce hit latency for the local processor”
Victim Replication Explained

Idea: keep data lines evicted from the private L1 cache in the local L2 cache slice

- Processor misses in the L2 cache, brings in a line from memory, gets placed in an L2 slice as normal
- Line also goes to L1 cache of that processor
- When this line is evicted from that L1 cache:
  - Invalidation or Writeback: do what we would normally do
  - Conflict or Capacity Miss: try to retain a copy in the local L2 slice
    - Global lines that are shared take precedence over victims
    - Replacement order: invalid line, global but unshared line, another replica
    - Never replicate a line on its home tile

Note: extra bits are required to keep track of this new information
In addition, the authors test a scheme called **Small Victim Cache (L2VC)**. This approach simply keeps an L1 victim cache on each tile where evicted lines go.
The authors make the following assumptions:

- L1I and L1D are kept small and private
- L2 slice is accessed with fixed latency and has quick tag resolution
- Latencies due to accessing remote L2 slices vary
- MESI protocol
## Simulation Parameters

<table>
<thead>
<tr>
<th>Component</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Model</td>
<td>in-order</td>
</tr>
<tr>
<td>Cache Line Size</td>
<td>64 B</td>
</tr>
<tr>
<td>L1 I-Cache Size/Associativity</td>
<td>16 KB/16-way</td>
</tr>
<tr>
<td>L1 D-Cache Size/Associativity</td>
<td>16 KB/16-way</td>
</tr>
<tr>
<td>L1 Load-to-Use Latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L1 Replacement Policy</td>
<td>Psuedo-LRU</td>
</tr>
<tr>
<td>L2 Cache Size/Associativity</td>
<td>1 MB/16-way</td>
</tr>
<tr>
<td>L2 Load-to-Use Latency (per slice)</td>
<td>6 cycles</td>
</tr>
<tr>
<td>L2 Replacement Policy</td>
<td>Random</td>
</tr>
<tr>
<td>L1 Victim Cache Size/Associativity</td>
<td>8 KB/16-way</td>
</tr>
<tr>
<td>L1 Victim Cache Load-to-Use Latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Network Configuration</td>
<td>4×2 Mesh</td>
</tr>
<tr>
<td>One-hop latency</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Worst case L2 hit latency (contention-free)</td>
<td>30 cycles</td>
</tr>
<tr>
<td>External memory latency</td>
<td>256 cycles</td>
</tr>
</tbody>
</table>
The authors use the 12 SpecINT2000 single-threaded bench marks and 8 multi-threaded OpenMP NAS Parallel Benchmarks.

**Benchmark Descriptions**

<table>
<thead>
<tr>
<th>Single-Threaded Benchmarks</th>
<th>Multi-Threaded Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Benchmark</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>bzip (3.8)</td>
<td>bzip2 compression algorithm</td>
</tr>
<tr>
<td>crafty (1.2)</td>
<td>High-performance chess program</td>
</tr>
<tr>
<td>eon (2.9)</td>
<td>Probabilistic ray tracer</td>
</tr>
<tr>
<td>gap (1.1)</td>
<td>A language used for computing in groups</td>
</tr>
<tr>
<td>gcc (6.4)</td>
<td>gcc compiler version 2.7.2.2</td>
</tr>
<tr>
<td>gzip (1.0)</td>
<td>Data compression using LZ77</td>
</tr>
<tr>
<td>mcf (1.7)</td>
<td>Single-depot vehicle scheduling algorithm</td>
</tr>
<tr>
<td>parser (5.6)</td>
<td>Word processing parser</td>
</tr>
<tr>
<td>perl-bmk (1.8)</td>
<td>Cut-down version of Perl v5.005.03</td>
</tr>
<tr>
<td>twolf (1.5)</td>
<td>The TimberWolfSC place/route package</td>
</tr>
<tr>
<td>vortex (1.5)</td>
<td>An object-oriented database program</td>
</tr>
<tr>
<td>vpr (5.3)</td>
<td>A FPGA place/route package</td>
</tr>
</tbody>
</table>
Results: Multi-threaded

The L2VR scheme has the lowest data access latency of all of the schemes, even though it has a higher off-chip miss rate than the L2S scheme.

(Lower is better)
Results: Multi-threaded

L2VR latency is lower because more L2 cache hits are local in the L2VR scheme, as seen below. In this figure, the 3 bars for each benchmark are L2P, L2S, L2VR, reading left to right.
L2VR essentially provides dynamically sized private L2 cache slice on the shared L2 cache slice.

**Figure:** % of L2$ for replicas

**Figure:** Coherence Traffic
Results: Single-threaded

Figure: % of L2$ for replicas by CPU

Figure: Cache Miss Rate
Results: Single-threaded

Figure: Average Access Latency

Figure: Coherence Messages
## Memory Access Latency: Comparing L2VR vs. L2S

<table>
<thead>
<tr>
<th>Single-Threaded Benchmarks</th>
<th>Multi-Threaded Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Benchmark</strong></td>
<td><strong>Reduction (%)</strong></td>
</tr>
<tr>
<td>bzip</td>
<td>18.5</td>
</tr>
<tr>
<td>crafty</td>
<td>27.9</td>
</tr>
<tr>
<td>eon</td>
<td>2.0</td>
</tr>
<tr>
<td>gap</td>
<td>13.5</td>
</tr>
<tr>
<td>gcc</td>
<td>21.1</td>
</tr>
<tr>
<td>gzip</td>
<td>46.4</td>
</tr>
<tr>
<td>mcf</td>
<td>41.4</td>
</tr>
<tr>
<td>parser</td>
<td>26.2</td>
</tr>
<tr>
<td>perl</td>
<td>5.1</td>
</tr>
<tr>
<td>twolf</td>
<td>36.1</td>
</tr>
<tr>
<td>vortex</td>
<td>15.3</td>
</tr>
<tr>
<td>vpr</td>
<td>30.1</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>23.7</strong></td>
</tr>
</tbody>
</table>
Elastic Cooperative Caching
Motivation

Applications being run have limited opportunity for parallelism
Machines often run many distinct applications simultaneously
Some applications reuse data often, others rarely do

Traditional approach is to either share the L2 cache or have private L2 caches

- **Shared**: an application that introduces a lot of new data can evict data that would be reused by another application
- **Private**: an application that reuses lots of data may have to go to main memory often if L2 cache is too small
Motivating Example

- Watching a YouTube video
- Using a word processor
- Using a web browser
- Antivirus software running
“Distributed cache partitioning mechanism that allows autonomous, scalable, and elastic cache behavior based on application needs”

- Distributed: each processor has local L2 cache
- Elastic: L2 caches are partitioned into shared and private sections
- Autonomous: decision to repartition is made independently by each processor

This allows applications that benefit from a larger L2 cache to utilize unused L2 cache space on other cores running applications that can have a smaller L2 cache
Keep the following possible utility types for applications in mind:

- **Saturating**: working set fits into cache, meaning increased cache size has no effect on the performance
Application Types

Keep the following possible utility types for applications in mind:

- **Saturating**: working set fits into cache, meaning increased cache size has no effect on the performance
- **Low**: use a lot of memory but do not reuse memory, meaning increased cache size does not improve performance
Application Types

Keep the following possible utility types for applications in mind:

- **Saturating**: working set fits into cache, meaning increased cache size has no effect on the performance
- **Low**: use a lot of memory but do not reuse memory, meaning increased cache size does not improve performance
- **Shared High**: lots of data sharing between threads
Keep the following possible utility types for applications in mind:

- **Saturating**: working set fits into cache, meaning increased cache size has no effect on the performance
- **Low**: use a lot of memory but do not reuse memory, meaning increased cache size does not improve performance
- **Shared High**: lots of data sharing between threads
- **Private High**: reuse data but do not share much between threads, meaning that increased cache sizes increase performance
Repartitioning

Elastic Cooperative Caching

Node 1 LRU LRU
L2
Private Shared
Hit Hit
Incr ++ Decr --
Counter
Cache Repartitioning Unit

Node 2 LRU LRU
L2
Private Shared
Hit Hit
Incr ++ Decr --
Counter
Cache Repartitioning Unit

Independent nodes, distributed structure
This whole approach relies on the ability for each node to logically define private and shared cache partitions.

- **Repartition Cycle (RC)**: repartitioning is done every set number of cycles
- **Counter**: decrements if least recently used way is in shared portion, increments otherwise
- **Upper Threshold (UT)**: if counter is above this value on an RC, then add a private way
- **Lower Threshold (LT)**: if counter is below this value on an RC, then add a shared way
Repartitioning

Other Notes:
- Thresholds are set at boot time
- Using thresholds prevents oscillatory states
If Private_LRU_Hit then
  Increase Counter
EndIf
If Shared_LRU_Hit then
  Decrease Counter
EndIf
If Repartition_Cycle then
  If Counter > Upper_Threshold then
    Add_Private_Way
    Send Repartition_Info_Msg
  ElseIf Counter < Lower_Threshold then
    Add_Shared_Way
    Send Repartition_Info_Msg
  EndIf
  Clear Counter
EndIf
Spilled Blocks

When a block is evicted, it is spilled to another node that has shared cache space available. Each node has a Spilled Block Allocator.

- This needs to be done efficiently
- With each repartition a message is sent to all nodes
- SBA uses this information to evict spilled blocks to nodes with larger shared caches
Round Robin Shared Way Selector
In the current setup, all applications would spill blocks. However, not all of them actually need to do this.

- We allow spilling when 75% of the cache is private
- We add a bit each entry in DCE that indicates shared. If shared, we allow it to spill

(If either one is true, a block is spilled)
ECC makes use of Distributed Coherence Engines in order to maintain scalability.

Each DCE is responsible for maintaining coherence for part of the address space.

If there is a local L1 and L2 miss, the request is sent to the corresponding DCE.

**Spilling**: If a cache block is the last on chip and is evicted, it lands on a neighboring tile (only allowed once per block).
### Experimental Setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Processors</td>
<td>16</td>
</tr>
<tr>
<td>Instr Window/ROB</td>
<td>16/48 entries</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>YAGS</td>
</tr>
<tr>
<td>Technology</td>
<td>70 nm</td>
</tr>
<tr>
<td>Frequency</td>
<td>4 GHz</td>
</tr>
<tr>
<td>Voltage</td>
<td>1.1 V</td>
</tr>
<tr>
<td>Block size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 I/D Cache</td>
<td>16 KB, 4-way</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB, 8-way</td>
</tr>
<tr>
<td>DCE Size</td>
<td>8192 entries</td>
</tr>
<tr>
<td>Network Type</td>
<td>Mesh with 2 VNC</td>
</tr>
<tr>
<td>Hop Latency</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Link BW</td>
<td>16 bytes/cycle</td>
</tr>
<tr>
<td>Memory Bus Latency</td>
<td>250 cycles</td>
</tr>
</tbody>
</table>

![Diagram of experimental setup]
- **Shared Memory**: Shared L2 cache distributed across the chip
**Configurations**

- **Shared Memory**: Shared L2 cache distributed across the chip
- **Private Memory**: Each processor has a private L2 cache
Configurations

- **Shared Memory**: Shared L2 cache distributed across the chip
- **Private Memory**: Each processor has a private L2 cache
- **Distributed Cooperative Caching**: Baseline configuration
Configurations

- **Shared Memory**: Shared L2 cache distributed across the chip
- **Private Memory**: Each processor has a private L2 cache
- **Distributed Cooperative Caching**: Baseline configuration
- **Adaptive Selective Replication**
Configurations

- **Shared Memory**: Shared L2 cache distributed across the chip
- **Private Memory**: Each processor has a private L2 cache
- **Distributed Cooperative Caching**: Baseline configuration
- **Adaptive Selective Replication**
- **Elastic Cooperative Cashing**: UT of 5, LT of 0 with RC every 100k cycles. Start with 4 private and 4 shared ways.
- **ECC + Adaptive Spilling**
Configurations

- **Shared Memory**: Shared L2 cache distributed across the chip
- **Private Memory**: Each processor has a private L2 cache
- **Distributed Cooperative Caching**: Baseline configuration
- **Adaptive Selective Replication**
- **Elastic Cooperative Cashing**: UT of 5, LT of 0 with RC every 100k cycles. Start with 4 private and 4 shared ways.
- **ECC + Adaptive Spilling**
- **Ideal**: Each node has 8 shared ways and 8 private ways, twice as many as before. No repartitioning. Provides an upper bound on ECC
Results

Ran workloads from the SPECOMP2001 in pairs, 8 threads each

Result Highlights for ECC

Performance
- 27% better than DCC
- 12% better than ASR
- 52% better than Private
- 53% better than Shared

Off-Chip Misses
- 18.6% reduction over DCC
- 16.4% reduction over ASR
Repartition Analysis

Average number of private ways for each pair of workloads as well as the percentage of blocks that were spilled in the ECC + AS as compared to ECC.
Repartition Analysis

Percentage of spilled blocks reused for different configurations
Repartition Analysis

Thread 1 of Equake application

![Graph showing miss rates and private cache sizes over time. The Y-axis represents miss rates (Fixed, ElasticCC) and the X-axis represents time (MCycles). There are labeled sections for Small Reuse, No Reuse, High Reuse, and Medium Reuse.}
Conclusion
L2VR and ECC better than L2S

- Both create dynamically sized private L2 slice (low latency), allow rest to be shared
- L2VR makes smart choices about what makes it to private slice (high reuse)
- ECC makes smart choices about where something is shared (low latency)
- ECC + AS makes smart choices about what makes it to shared space (low latency, high reuse)
Questions?