Previously in CS211: Sectored Caches

• Divide a block into subblocks (or sectors)
  • Have separate valid and dirty bits for each sector
  • When is this useful? (Think writes...)
  • How many subblocks do you transfer on a read?

++ No need to transfer the entire cache block into the cache
   (A write simply validates and updates a subblock)
++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully)

-- More complex design
-- May not exploit spatial locality fully when used for reads
Previously in CS211: Instruction vs. Data Caches

• Unified:
  + Dynamic sharing of cache space between instructions and data
  -- Instructions and data can thrash each other (i.e., no guaranteed space for either)
  -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

• First level caches are almost always split
  • Mainly for the last reason above

• Second and higher levels are almost always unified
Previously in CS211: Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity
    - Tag store and data store accessed in parallel
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
    - Tag store and data store accessed serially

- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter
Case Study: ARM Cortex-A53 Cache Systems

- L1 I-Cache is 8KB to 64 KB, has 64B cache lines, is 2-way set associative, and has a 128-bit read interface to L2
- L1 D-Cache is 8KB to 64 KB, has 64B cache lines, is 4-way set associative, has a 128-bit read interface to L2, and a 256-bit write interface to L2
- L2 Cache is 128KB to 2 MB, has 64B cache lines, and is 16-way set associative
- Both the L1 D cache and L2 use a write-back policy defaulting to allocate on write.
- LRU approximation in all the caches
Case Study: Intel Core i7 6700

- L1 I-Cache is 32KB, has 64B cache lines, is 8-way set associative
- L1 D-Cache 32KB, has 64B cache lines, is 8-way set associative
- L1 I-Cache and D-Cache have Pseudo-LRU replacement
- L2 Cache is 256KB, has 64B cache lines, is 4-way set associative
- L2 Cache has Pseudo-LRU replacement
- L3 Cache is 8MB, 2MB per core, has 64B cache lines, is 16-way set associative
- L3 Cache has Pseudo-LRU replacement but with an ordered selection algorithm
  - The block replaced is always the lowest numbered way whose access bit is off
Cache Inclusion Policy
Inclusive

Initial state

L1: Read A miss; load A into L1 and L2
L2: Read B miss; load B into L1 and L2

Evict A from L1 due to cache replacement
Evict B from L2 due to cache replacement

Back invalidation
Exclusive

Initial state

Read A miss; load A into L1

Read B miss; load B into L1

Evict A from L1 due to cache replacement and place in L2
Non-inclusive

L1

A

A

A

B

B

Initial state

Read A miss; load A into L1 and L2

Read B miss; load B into L1 and L2

Evict A from L1 due to cache replacement

Evict B from L2 due to cache replacement
Cache Inclusion Policy

• Multi-level caches are designed depending upon if data in one cache level are also in other cache levels

• Inclusive Policy
  • Same data in all levels

• Exclusive Policy
  • Data in only one cache

• Exclusive policy increases effective amount of caching, but:
  • If data in L2 but not L1, then block is moved from L2 to L1
  • If this causes an eviction from L1, then victim cache block moved to L2

• Non-inclusive policy is a blend of inclusive and exclusive policies
Inclusive, or not?

• Inclusive cache eases coherence
  • Updating a cache block in L1 entails an update in inclusive LLC.
  • A non-inclusive LLC, say L2 cache, which needs to evict a block, must ask L1 cache if it has the block, because such information is not present in LLC.

• Non-inclusive cache yields higher performance though, why?
  • No back invalidation
  • More data can be cached
‘Sneaky’ LRU for Inclusive Cache

A is frequently used in L1 cache. It is MRU in L1 cache.

A is frequently hit in L1 cache. It is MRU in L1 cache.

A is evicted for replacement, in both L1 and L2.

In LLC, A is LRU.

In LLC, A is not frequently hit.

As a result, MRU block that should be retained might be evicted, which causes performance penalty.

What if LLC is non-inclusive?

Link: https://doi.org/10.1109/MICRO.2010.52
Main Memory
Modern Virtual Memory Systems

*Illusion of a large, private, uniform store*

**Protection & Privacy**
several users, each with their private address space and one or more shared address spaces

**Demand Paging**
Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations

*The price is address translation on each memory reference*
Recap: Hierarchical Page Table

Virtual Address from CPU

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>21</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td>offset</td>
</tr>
</tbody>
</table>

10-bit L1 index
10-bit L2 index

Root of Current Page Table

(Processor Register, satp in RISC-V)

Level 1 Page Table

Level 2 Page Tables

Data Pages

Physical Memory

Page in primary memory
Page in secondary memory
PTE of a nonexistent page

RISC-V Sv32 Virtual Memory Scheme
Page-Based Virtual-Memory Machine
(Hardware Page-Table Walk)

- Assumes page tables held in untranslated physical memory

*Page Fault? Protection violation?*

Virtual Address

• Inst. TLB
  • Inst. Cache
  • Decode
  • E
  • Data TLB
  • Data Cache
  • W

Physical address used to access cache

• Page-Table Base Register
• Hardware Page Table Walker

• Physical address used to access cache

Memory Controller

Main Memory (DRAM)
The Main Memory System

- Main memory is a critical component of all computing systems: server, mobile, embedded, desktop, sensor

In-memory database  Social networks  In-memory analytics  Data centers
The Main Memory System

- Main memory is a critical component of all computing systems: server, mobile, embedded, desktop, sensor

- Main memory system must scale (in size, technology, efficiency, cost, and management algorithms) to maintain performance growth and technology scaling benefits
Main Memory

• **Major Trends Affecting Main Memory**

• The Memory Scaling Problem and Solution Directions
  • New Memory Architectures
  • Enabling Emerging Technologies

• How Can We Do Better?
Major Trends Affecting Main Memory (II)

• Need for main memory capacity, bandwidth, QoS increasing
  • Multi-core: increasing number of cores
  • Data-intensive applications: increasing demand/hunger for data
  • Consolidation: cloud computing, GPUs, mobile, heterogeneity

• Main memory energy/power is a key system design concern

• DRAM technology scaling is ending
Example: The Memory Capacity Gap

Core count doubling ~ every 2 years
DRAM DIMM capacity doubling ~ every 3 years

- Memory capacity per core expected to drop by 30% every two years
- Trends worse for memory bandwidth per core!
Major Trends Affecting Main Memory (III)

- Need for main memory capacity, bandwidth, QoS increasing

- Main memory energy/power is a key system design concern
  - ~40-50% energy spent in off-chip memory hierarchy [Lefurgy, IEEE Computer 2003]
  - DRAM consumes power even when not used (periodic refresh)

- DRAM technology scaling is ending
Major Trends Affecting Main Memory (IV)

- Need for main memory capacity, bandwidth, QoS increasing
- Main memory energy/power is a key system design concern
- DRAM technology scaling is ending
  - ITRS projects DRAM will not scale easily below X nm
  - Scaling has provided many benefits:
    - higher capacity (density), lower cost, lower energy
Main Memory

• Major Trends Affecting Main Memory
• The Memory Scaling Problem and Solution Directions
  • New Memory Architectures
  • Enabling Emerging Technologies
• How Can We Do Better?
The DRAM Scaling Problem

- DRAM stores charge in a capacitor (charge-based memory)
  - Capacitor must be large enough for reliable sensing
  - Access transistor should be large enough for low leakage and high retention time
  - Scaling beyond 40-35nm (2013) is challenging [ITRS, 2009]

- DRAM capacity, cost, and energy/power hard to scale
Solution 1: Fix DRAM

• Overcome DRAM shortcomings with
  • System-DRAM co-design
  • Novel DRAM architectures, interface, functions
  • Better waste management (efficient utilization)

• Key issues to tackle
  • Enable reliability at low cost
  • Reduce energy
  • Improve latency and bandwidth
  • Reduce waste (capacity, bandwidth, latency)
  • Enable computation close to data
Solution 1: Fix DRAM

- Avoid DRAM:
Solution 2: Emerging Memory Technologies

• Some emerging resistive memory technologies seem more scalable than DRAM (and they are non-volatile)

• Example 1: Phase Change Memory
  • Expected to scale to 9nm (2022 [ITRS])
  • Expected to be denser than DRAM: can store multiple bits/cell

• Example 2: Intel Optane DC Memory
  • Commercially available, in terabytes

• But, emerging technologies have shortcomings as well
  • Can they be enabled to replace/augment/surpass DRAM?

  ▪ Zhao+, “FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems,” MICRO 2014.
  ▪ ...
Solution 3: Hybrid Memory Systems

Hardware/software manage data allocation and movement to achieve the best of multiple technologies

Yoon, Meza et al., “Row Buffer Locality Aware Caching Policies for Hybrid Memories,” ICCD 2012 Best Paper Award.
Some Promising Directions

• New memory architectures
  • Rethinking DRAM
  • A lot of hope in fixing DRAM

• Enabling emerging NVM technologies
  • Hybrid memory systems
  • Single-level memory and storage
  • A lot of hope in hybrid memory systems and single-level stores
Virtual Memory and Cache Interaction
Cache-VM Interaction

- Physical cache
- Virtual (L1) cache
- Virtual-physical cache
Address Translation and Caching

- When do we do the address translation?
  - Before or after accessing the L1 cache?

- In other words, is the cache virtually addressed or physically addressed?
  - Virtual versus physical cache

- What are the issues with a virtually addressed cache?

  - **Synonym problem:**
    - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data
Homonyms and Synonyms

• **Homonym: Same VA can map to two different PAs**
  • Why?
    • VA is in different processes

• **Synonym: Different VAs can map to the same PA**
  • Why?
    • Different pages can share the same physical frame within or across processes
    • Reasons: shared libraries, shared data, copy-on-write pages within the same process, ...

• Do homonyms and synonyms create problems when we have a cache?
  • Is the cache virtually or physically addressed?
Virtually-Indexed Physically-Tagged

• If $C \leq (\text{page\_size} \times \text{associativity})$, the cache index bits come only from page offset (same in VA and PA)

• If both cache and TLB are on chip
  • index both arrays concurrently using VA bits
  • check cache tag (physical) against TLB output at the end
Virtually-Indexed Physically-Tagged

- If $C > (\text{page\_size} \times \text{associativity})$, the cache index bits include VPN $\Rightarrow$
  
  **Synonyms can cause problems**
  - Different VAs mapped to the same physical address
  - The same physical address can exist in two locations

- Solutions?
Some Solutions to the Synonym Problem

• Limit cache size to \((\text{page size} \times \text{associativity})\)
  • get index from page offset

• On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  • Used in Alpha 21264, MIPS R10K

• Restrict page placement in OS
  • make sure \(\text{index(VA)} = \text{index(PA)}\)
  • Called page coloring
  • Used in many SPARC processors
PIPT and VIVT

- Physically indexed, physically tagged (PIPT) caches use the physical address for both the index and the tag.
  - Simple to implement but slow, as the physical address must be looked up (which could involve a TLB miss and access to main memory) before that address can be looked up in the cache.

- Virtually indexed, virtually tagged (VIVT) caches use the virtual address for both the index and the tag.
  - Potentially much faster lookups.
  - Problems when several different virtual addresses may refer to the same physical address
    - Addresses would be cached separately despite referring to the same memory, causing coherency problems.
    - Additionally, there is a problem that virtual-to-physical mappings can change, which would require clearing cache blocks

- Can we have PIVT?
Conclusion

• Case studies for cache
• Cache inclusion
• Main memory
• Interaction between cache and memory
Acknowledgements

• These slides contain materials developed and copyright by:
  • Prof. Onur Mutlu (ETH Zurich)
  • Prof. Zhi Wang (FSU)
  • Prof. Jason Tang (UMBC)
  • Prof. Krste Asanović (UC Berkeley)