Multi-Core Memory Hierarchies
Lecture 2: Overview

Tong Dong

http://mprc.pku.edu.cn/~tongdong/MMH
In last lecture

- Memory Wall!
- Why Multi-core? Power Wall!
- Trends in Technologies: Moore’s Law is slowing down; the end of Dennard (CMOS) scaling.
- Accelerator Era. Energy-Efficiency First!
- Energy Efficiency: GFLOS/Watts, Ops/J
  - High energy efficiency not equate high performance
  - Need to mitigate useless energy consumption
- Important Merits and Analyst Tools
- Quantitative Principles

*It’s Memory, Stupid !!!*
Heterogeneous Multicore

- More energy efficient processor with large caches
  - Complex Core: OoO Superscalar + narrow SIMD
  - Simple Core: in-order processor
  - General-purpose Accelerator: wide SIMD, GPU
  - Hard-wired Custom Accelerator: for long-time standard
  - Reconfigurable Accelerator (FPGA): for variant standard
Dark Silicon

- Heterogeneous Multicore under power budget
  - DVFS: Dynamic Frequency/Voltage Scaling
  - Dark silicon/Ultra-low Voltage

Power-down when needless
Technology Potential in Memory System Hierarchy

- SSD pushing HDD into cold data / archival role
- Latency gap between DRAM and SSD
- Fill this gap with a non-volatile component
- Persistent Memory – a new class of NVM (non-volatile memory)
Memory-Wall & Power-Wall: 3D-Stack

Experimental demonstrations:
(a) 3D RRAM
(b) Efficient heat removal solutions
(c) Monolithic 3D "high-rise chip"

(1) Energy-efficient FETs
- 1D CNTs
- 2D layered nanomaterials

(2) High-density nonvolatile memories
- 3D RRAM: massive storage
- STT-MRAM: quick access

(3) Fine-grained monolithic 3D integration
- Compute + memory elements
- Ultradense connectivity using nanoscale vias

(4) Efficient heat removal

(5) Computation immersed in memory

On-chip nanoconvection/conduction solutions
Performance Model for Multi-Core

**Multi-Core (CPU) vs. Multi-Thread (GPGPU)**

\[-Perf = \min(N \times \frac{freq}{CPI_{exe}}) \times \min \left(1, \frac{T}{1+t\frac{r_m}{CPI_{exe}}}, \frac{BW_{max}}{r_m \times m_{L1} \times b}\right)\]

- \( t = (1 - m_{L1}) t_{L1} + m_{L1}(1 - m_{L2}) t_{L2} + m_{L1} m_{L2} t_{mem}\)
- \( m_{L1} = \left(\frac{c_{L1}}{T\beta_{L1}}\right)^{1-\alpha_{L1}}, m_{L2} = \left(\frac{c_{L2}}{T\beta_{L2}}\right)^{1-\alpha_{L2}}\)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
<th>Impacted By</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>Number of cores</td>
<td>4</td>
<td>Multicore Topology</td>
</tr>
<tr>
<td>(T)</td>
<td>Number of threads per core</td>
<td>1</td>
<td>Core Style</td>
</tr>
<tr>
<td>(freq)</td>
<td>Core frequency (MHz)</td>
<td>3200</td>
<td>Core Performance</td>
</tr>
<tr>
<td>(CPI_{exe})</td>
<td>Cycles per instruction (zero-latency cache accesses)</td>
<td>1</td>
<td>Core Performance, Application</td>
</tr>
<tr>
<td>(C_{L1})</td>
<td>L1 cache size per core (KB)</td>
<td>64</td>
<td>Core Style</td>
</tr>
<tr>
<td>(C_{L2})</td>
<td>L2 cache size per chip (MB)</td>
<td>2</td>
<td>Core Style, Multicore Topology</td>
</tr>
<tr>
<td>(t_{L1})</td>
<td>L1 access time (cycles)</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>(t_{L2})</td>
<td>L2 access time (cycles)</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>(t_{mem})</td>
<td>Memory access time (cycles)</td>
<td>426</td>
<td>Core Performance</td>
</tr>
<tr>
<td>(BW_{max})</td>
<td>Maximum memory bandwidth (GB/s)</td>
<td>200</td>
<td>Technology Node</td>
</tr>
<tr>
<td>(b)</td>
<td>Bytes per memory access (B)</td>
<td>64</td>
<td>-</td>
</tr>
<tr>
<td>(f)</td>
<td>Fraction of code that can be parallel</td>
<td>varies</td>
<td>Application</td>
</tr>
<tr>
<td>(r_m)</td>
<td>Fraction of instructions that are memory accesses</td>
<td>varies</td>
<td>Application</td>
</tr>
<tr>
<td>(a_{L1}, \beta_{L1})</td>
<td>L1 cache miss rate function constants</td>
<td>varies</td>
<td>Application</td>
</tr>
<tr>
<td>(a_{L2}, \beta_{L2})</td>
<td>L2 cache miss rate function constants</td>
<td>varies</td>
<td>Application</td>
</tr>
</tbody>
</table>

Performance Model for Multi-Core

Fig. 3. Performance for different compute/memory ratios.

Fig. 2. Performance for different cache hit rate functions.

Fig. 4. Performance for different off-chip latencies.

Dark Silicon

Figure 4: Amdahl’s law projections for the dynamic topology. Upperbound of all four topologies (x-axis: technology node).
Dark Silicon Era

- 80% dark silicon in 8-nm and 50% dark silicon in 16-nm technology node
- Not materialized in 22- and 16-nm
- Thermal Design Power (TDP) constraint

- The deployment of FinFETs and DVFS lead to new less-conservative predictions.
- 50% dark silicon in 8-nm and 30% in low-power mode under temperate constrains

---

**Esmaeilzadeh@ISCA’11**

- Device Scaling (DevM) × Core Scaling (CorM) × Multicore Scaling (CmpM) => Optimal # of Cores
- Microarchitectural Features
- Application Behavior
- 80% dark silicon in 8-nm and 50% dark silicon in 16-nm technology node
- Not materialized in 22- and 16-nm
- Thermal Design Power (TDP) constraint

**Henkel, shafique@DAC’15**

- Pattern(a): $T_{DTM}$ is exceeded
  - 52 cores @3.6 GHz, $P_{total}=196$ W
- Pattern(b): $T_{DTM}$ is not exceeded
  - 60 cores@3.6 GHz, $P_{total}=226$ W
Model of cache performance

- For N cores and T thread per core, the miss ratio of the $i$-level cache:
  - This model assumes that all threads are independent of each other, and that thread context is saved in the cache (or in other dedicated on-chip storage) when threads swap out.

$$m_{Li} = \left( \frac{C_{Li}}{NT \beta_{Li} I} \right)^{1-\alpha_{Li}}$$

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

- Average Access Time in $i$-level cache:

$$t_{Li} = (1 - m_{Li})t_{Li}^{hit} + m_{Li} t_{Li-1}$$
Performance Merits in Multi-Core

- **Multi-Core Speedup: System Throughput**
  \[ \text{WeightedSpeedup} = \sum_i \frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{alone}}} \]

- **Multi-Core Fairness: maximum slowdown**
  \[ \text{Slowndown} = \frac{\text{IPC}^{\text{alone}}}{\text{IPC}^{\text{shared}}} \]
  - Min-Max Fairness: Slowdown Equality

- **Harmonic Mean: balanced performance and fairness**
  \[ \text{HarmonicSpeedup} = n / \left( \frac{1}{s_1} + \frac{1}{s_2} + \cdots + \frac{1}{s_n} \right) \]

- **QoS: Quality-of-Service**
  - Deadline; Avoiding Starvation
  - Long Tail in WSC: 90%~95%

- **Latency vs. bandwidth**
“It’s interference, stupid!”: Capacity

Locality of reference: temporal locality and spatial locality
“It’s Interference, Stupid!”: Capacity

Thread1

Core

Shared$

Thread2

Core

Shared$

alone

Thread1

Core1

Thread2

Core2

shared

Thrashing
Interference in Shared Cache

**Figure 1.** The performance degradation relative to running solo for two different schedules of SPEC CPU2006 applications on an Intel Xeon X3565 quad-core processor (two cores share an LLC).
Interference Model for Shared Cache

- **Cache Sensitivity**
  - A Measure of how much an application will suffer when cache space is taken away from it due to contention.
  - \[ S = \left(\frac{1}{1+n}\right) \sum_{i=0}^{n} i \ast h(i) \]
  - Where \( h(i) \) is the hit number to the i-th position in LRU stack, \( n \) is the size of the stack.

- **Cache Intensity**
  - A measure of how much an application will hurt others by taking away their space in a shared cache.
  - The number of last-level cache access per one million instructions, RPMI
Interference Model for Shared Cache

- Combining sensitivity S and intensity Z into the Pain metric
  - Used to approximate the co-run degradations

- Pain of A due to B
  - $Pain(A|B) = S(A) \times Z(B)$

- The degradation of co-scheduling A and B together
  - $Pain(A, B) = Pain(A|B) + Pain(B|A)$
“It’s interference, stupid!”: Queue Time

Locality of reference: temporal locality and spatial locality
It’s Interference, Stupid! – Shared Cache

Thread1

Core

Thread2 alone

Core

-----------------------------

Thread1

Core1

Thread2

Core2

Shared

Latency Increased
Little’s Law

- Mean arrival rate: $\lambda$, Mean service rate: $\mu$, Utilization: $\rho = \lambda/\mu$
- Poisson arrivals and Exponential Service: $\lambda = \sum_{i=0}^{n} \lambda_i$
- Little’s law: The average number of customers in the system, $L$, is equal to the average arrival rate of customer to the system, multiplied by the average system time per customer, $W$. 
  \[ L = \lambda W \]
A Little Queue Theory

- Mean Number in System: $L = \frac{\lambda}{\mu-\lambda} = \frac{\rho}{1-\rho}$
- Variance of Number in System: $Var = \frac{\rho}{(1-\rho)^2}$
- Mean Queue Length: $L_q = L - \rho = \frac{\rho^2}{1-\rho}$
- **Average Response Time**: $R = \frac{1}{\mu-\lambda} = \frac{1}{\mu(1-\rho)}$
- Average Waiting Time: $W_q = \frac{\rho}{\mu-\lambda} = \frac{\rho}{\mu(1-\rho)}$
- Average size of queue when it is not empty:
  \[ L'_q = \frac{\mu}{\mu - \lambda} = \frac{1}{1 - \rho} \]
- Throughput, Utilization, and Traffic Intensity:
  \[ U = \frac{\lambda}{\mu} \]
Latency Interference in Queue Theory

- $\text{Slowdown}(a\mid b) = \frac{\text{Performance}_a}{\text{Performance}_{a,b}} = \frac{\mu(1-\rho_a-\rho_b)}{\mu(1-\rho_a)} = 1 - \frac{\rho_b}{1-\rho_a} = 1 - \frac{\lambda_b}{\mu-\lambda_a}$

- If there are $n$ sub-flows in flow $b$ and all is Poisson Distribution, then

$$\text{slowdown}(a) = 1 - \frac{\lambda_1}{\mu-\lambda_a} - \frac{\lambda_2}{\mu-\lambda_a} - \cdots - \frac{\lambda_n}{\mu-\lambda_a} = 1 - \sum_{i=1}^{n} \text{Interf}(a\mid i)$$
Latency: Scheduling Algorithm
- FCFS (first come, first served)
- LCFS (last come, first served)
- LCFS-PR (last come, first served, preempt resume)
- SIRO (service in random order)
- RR (round robin)
- PS (processor sharing)
- IS (infinite server)
- PRIIO (priority scheduling for multi-customer)

Bandwidth: Weighted Bandwidth Allocation

\[ BW(i) = \rho' \mu \cdot \left( \frac{W_i}{W_1 + W_2 + \cdots + W_n} \right) \]
Topics on Cache/Main Memory/Storage

- **Cache**
  - Replacement
  - Prefetch
  - Partition
  - Compression Cache
  - 3D DRAM Cache

- **Main Memory**
  - DRAM Scheduling/Partition/Mapping
  - Bandwidth Partition/Throttling

- **Hot Topics**
  - GPU/Accelerator
  - Security
  - Scratchpad, Software-managed cache
  - ......
Cache in everywhere

- Storage systems
- Databases
- Web servers
- Middleware
- Processors
- File systems
- Disk drives
- RAID controllers
- Operating systems
- In applications: Compression, list updating, etc.
Cache Management in Memory Paging

- Offline Optimal: MIN [Belady, 1966]
- Recency
  - LRU/CLOCK
  - LRU Stack Algorithm [Mattson, 1970], MRC
- Frequency
  - LFU, LRU-2, 2Q, MQ, LIRS
- Recency and Frequency:
  - FBR, LRFU, ALRFU
- ARC, CAR, CLOCK-Pro, ...
- Web Cache: Greedy Dual, …, Greedy-Dual-Size-Frequency

\[ Pr(f) = Clock + Fr(f) \times \frac{Cost(f)}{Size(f)} \]
Set Associate Cache in processor

- **N-way set associative**: N entries for each Cache Index
  - N direct mapped caches operates in parallel

- **Example**: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared to the input in parallel
  - Data is selected based on the tag result
Review: Four Questions for Memory Hierarchy Designers

- Q1: Where can a block be placed in the upper level?  
  *(Block placement)*  
  - Fully Associative, Set Associative, Direct Mapped

- Q2: How is a block found if it is in the upper level?  
  *(Block identification)*  
  - Tag/Block

- Q3: Which block should be replaced on a miss?  
  *(Block replacement)*  
  - Random, LRU

- Q4: What happens on a write?  
  *(Write strategy)*  
  - Write Back or Write Through (with Write Buffer)
Reducing cache miss

- **Classifying Misses: 3 Cs**
  - **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. Also called cold start misses or first reference misses. *(Misses in even an Infinite Cache)*
  - **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. *(Misses in Fully Associative Size X Cache)*
  - **Conflict**—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses. *(Misses in N-way Associative, Size X Cache)*

- **More recent, 4th “C”:**
  - **Coherence** - Misses caused by cache coherence.
# Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Hit time</th>
<th>Bandwidth</th>
<th>Miss penalty</th>
<th>Miss rate</th>
<th>Power consumption</th>
<th>Hardware cost/complexity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small and simple caches</td>
<td>+</td>
<td></td>
<td>–</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>Trivial; widely used</td>
</tr>
<tr>
<td>Way-predicting caches</td>
<td>+</td>
<td></td>
<td></td>
<td>+</td>
<td>1</td>
<td>1</td>
<td>Used in Pentium 4</td>
</tr>
<tr>
<td>Pipelined cache access</td>
<td>–</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>Widely used</td>
</tr>
<tr>
<td>Nonblocking caches</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>Widely used</td>
</tr>
<tr>
<td>Banked caches</td>
<td>+</td>
<td></td>
<td></td>
<td>+</td>
<td>1</td>
<td>1</td>
<td>Used in L2 of both i7 and Cortex-A8</td>
</tr>
<tr>
<td>Critical word first and early restart</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>Widely used</td>
</tr>
<tr>
<td>Merging write buffer</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>Widely used with write through</td>
</tr>
<tr>
<td>Compiler techniques to reduce cache misses</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>Software is a challenge, but many compilers handle common linear algebra calculations</td>
</tr>
<tr>
<td>Hardware prefetching of instructions and data</td>
<td>+</td>
<td>+</td>
<td>–</td>
<td></td>
<td>2 instr., 3 data</td>
<td>Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.</td>
<td></td>
</tr>
<tr>
<td>Compiler-controlled prefetching</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>Needs nonblocking cache; possible instruction overhead; in many CPUs</td>
</tr>
</tbody>
</table>

**Figure 2.11** Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.
## Cache Placement

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Placement</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Anywhere; Int, FP, SPR</td>
<td>Compiler/programmer manages</td>
</tr>
<tr>
<td>Cache (SRAM)</td>
<td>Fixed in H/W</td>
<td>Direct-mapped, set-associative, fully-associative</td>
</tr>
<tr>
<td>DRAM</td>
<td>Anywhere</td>
<td>O/S manages</td>
</tr>
<tr>
<td>Disk</td>
<td>Anywhere</td>
<td>O/S manages</td>
</tr>
</tbody>
</table>
Cache Replacement

- How do we choose *victim*?
  - Verbs: *Victimize, evict, replace, cast out*

- Many policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used), pseudo-LRU
  - LFU (least frequently used)
  - NMRU (not most recently used)
  - NRU
  - Pseudo-random (yes, really!)
  - Optimal
  - etc.
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 look-ahead (Cheetah simulator)

- Useful, since it reveals opportunity
  - (X,A,B,C,D,X): LRU 4-way SA $, 2^{nd} X will miss
Last-Recently Used: LRU-a

- For $a=2$, LRU is equivalent to NMRU
  - Single bit per set indicates LRU/MRU
  - Set/clear on each access
- For $a>2$, LRU is difficult/expensive
  - Timestamps? How many bits?
    - Must find min timestamp on each eviction
  - Sorted list? Re-sort on every access?
- List overhead: $\log_2(a)$ bits/block
  - Shift register implementation
Practical Pseudo-LRU

- Rather than true LRU, use binary tree
- Each node records which half is older/newer
- Update nodes on each reference
- Follow older pointers to find LRU victim
Practical Pseudo-LRU In Action

Partial Order Encoded in Tree:

\[
\begin{align*}
Z &< A \\
Y &< X \\
B &< C \\
J &< F \\
A &> X \\
C &< F \\
A &> F
\end{align*}
\]
Binary tree encodes PLRU *partial order*

- At each level point to LRU half of subtree

Each access: flip nodes along path to block

Eviction: follow LRU path

Overhead: \((a-1)/a\) bits per block

Refs: J,Y,X,Z,B,C,F,A

011: PLRU
Block B is here

110: MRU
block is here
True LRU Shortcomings

- Streaming data/scans: \( x_0, x_1, \ldots, x_n \)
  - Effectively no temporal reuse

- Thrashing: reuse distance > \( a \)
  - Temporal reuse exists but LRU fails

- All blocks march from MRU to LRU
  - Other conflicting blocks are pushed out

- For \( n>a \) no blocks remain after scan/thrash
  - Incur many conflict misses after scan ends

- Pseudo-LRU sometimes helps a little bit
Not Recently Used (NRU)

- Keep NRU state in 1 bit/block
  - Bit is set to 0 when installed (assume reuse)
  - Bit is set to 0 when referenced (reuse observed)
  - Evictions favor NRU=1 blocks
  - If all blocks are NRU=0
    - Eviction forces all blocks in set to NRU=1
    - Picks one as victim (pseudo-random, rotating, fixed left-to-right)

- Simple, similar to virtual memory clock algorithm
- Provides some scan and thrash resistance
  - Relies on “randomizing” evictions rather than strict LRU order
- Used by Intel Itanium, Sparc T2
Least Frequently Used

- Counter per block, incremented on reference
- Evictions choose lowest count
  - Logic not trivial ($a^2$ comparison/sort)
- Storage overhead
  - 1 bit per block: same as NRU
  - How many bits are helpful?
Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?
Cache Design in Multi-Core Era

- Large Cache Design
- Shared vs Private Cache
- Centralized and Distributed Cache
- NUCA
- Replacement
- Partition/Prefetching
- Compression/3D-DRAM Cache/eDRAM
- Cache Coherence
- NoC
- NVM
Shared Caches
Private Cache
Shared vs. Private LLC

**Shared LLC**
- No replication of blocks
- Dynamic allocation of space among cores
- Low latency for shared data in LLC (no indirection thru directory)
- No interconnect traffic or tag replication to maintain directories

**Private LLC**
- More isolation and better quality-of-service
- Lower wire traversal when accessing LLC hits, on average
- Lower contention when accessing some shared data
- No need for software support to maintain data proximity
## Shared vs. Private LLC

<table>
<thead>
<tr>
<th>Shared LLC Cache</th>
<th>Private LLC Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No Replication of shared blocks</strong> (higher effective capacity)</td>
<td><strong>Replication of shared blocks</strong> (lower effective capacity)</td>
</tr>
<tr>
<td><strong>Dynamic allocation of Space</strong> (higher effective capacity)</td>
<td><strong>Design-time allocation of space among cores</strong> (low effective capacity)</td>
</tr>
<tr>
<td><strong>Quick traversal through coherence interface</strong> (low latency for shared data)</td>
<td><strong>Slower traversal through coherence interface</strong> (high latency for shared data)</td>
</tr>
<tr>
<td><strong>No LLC tag replication for directory implementation</strong> (low area requirement)</td>
<td><strong>Directory implementation requires replicated LLC tags</strong> (high area requirements)</td>
</tr>
<tr>
<td><strong>Higher interference between threads</strong> (negatively impacts QoS)</td>
<td><strong>No interference between threads</strong> (positively impacts QoS)</td>
</tr>
<tr>
<td><strong>Longer wire traversals on average to detect an LLC Hit</strong> (high average hit latency)</td>
<td><strong>Short wire traversals on average to detect an LLC hit</strong> (low average hit latency)</td>
</tr>
<tr>
<td><strong>High contention when accessing shared resource</strong> (high hit latency for private data)</td>
<td><strong>No Contention when accessing LLC cache</strong> (low hit latency for private data)</td>
</tr>
</tbody>
</table>
Clustered LLC
Centralized LLC

Figure 1.5: Shared L2 cache with a centralized layout (the L2 cache occupies a contiguous area in the middle of the chip and is surrounded by cores). An on-chip network is required to connect the many cache banks to each other and the cores. In all these cases, the functionality of the cache controller is replicated in each of the banks to avoid having to go through a central entity.
Distributed Shared LLC

A single tile composed of a core, L1 caches, and a bank (slice) of the shared L2 cache

The cache controller forwards address requests to the appropriate L2 bank and handles coherence operations

Memory Controller for off-chip access
Inclusion

LAP: Loop-Block Aware Inclusion Properties for Energy-Efficient Asymmetric Last Level Caches, ISCA'16

(a) Non-inclusive
- No redundant clean insertion
- Redundant LLC data-fill
- Useless duplicate data

(b) LAP
- Insert only non-duplicate data
- Duplicate only useful clean data
- No redundant LLC data-fill

(c) Exclusive
- Redundant clean insertion
- No duplicate data
- No redundant LLC data-fill

✓: pros  ➢: cons

No-invalidation on L3 hits & loop-block-aware replacement policy

<table>
<thead>
<tr>
<th>In L3?</th>
<th>L2</th>
<th>L3 hit</th>
<th>Clean victim</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Invalidate on hit</th>
<th>L3 fill</th>
<th>Clean Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>noni</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ex</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>LAP</td>
<td>N</td>
<td>N</td>
<td>Y if not in L3</td>
</tr>
</tbody>
</table>
Summary

- Is Interference, stupid!
  - Capacity Interference
  - Queue Interference

- Cache Basics

- Last-Level Cache (LLC)
  - Shared vs. Private
  - Centralized vs. Distributed
  - Inclusive vs. exclusive vs. non-inclusive caches