Mo Money, No Problems: Caches #2...
Reminder: Cache Terms...

- **Cache**: A small and fast memory used to increase the performance of accessing a big and slow memory
- Uses **temporal locality**: The tendency to reuse data in the same space over time
- Uses **spacial locality**: The tendency to use data at addresses near
- Cache **hit**: The address being fetched is in the cache
- Cache **miss**: The address being fetched is not in the cache
- **Valid bit**: Is a particular entry valid
- Cache **flush**: Invalidate all entries
CPU-Cache Interaction (5-stage pipeline)

Cache Refill Data from Lower Levels of Memory Hierarchy

Stall entire CPU on data cache miss

To Memory Control

wdata hit?

L1 D$ rdata

wdata

ALU

MD1

MD2

Decode, Register Fetch

E

A

B

M

Y

ALU

Y

MD1

MD2

wdata hit?

L1 D$

we

addr

rdata

wdata

Stall entire CPU on data cache miss

PCen

PC

addr

inst

hit?

bubble

0x4

Add

C

wdata

Cache Refill Data from Lower Levels of Memory Hierarchy

To Memory Control

wdata

hit?

L1 I$

E

A

B

M

Y

ALU

Y

MD1

MD2

wdata hit?

L1 D$

we

addr

rdata

wdata

Stall entire CPU on data cache miss

PCen

PC

addr

inst

hit?

bubble

0x4

Add

C

wdata

Cache Refill Data from Lower Levels of Memory Hierarchy

To Memory Control

wdata

hit?

L1 I$

E

A

B

M

Y

ALU

Y

MD1

MD2

wdata hit?

L1 D$

we

addr

rdata

wdata

Stall entire CPU on data cache miss

PCen

PC

addr

inst

hit?

bubble

0x4

Add

C

wdata

Cache Refill Data from Lower Levels of Memory Hierarchy

To Memory Control

wdata

hit?

L1 I$
More Terms

- **Cache level:**
  - The order in the memory hierarchy: L1$ is closest to the processor
  - L1 caches may only hold data (Data-cache, D$) or instructions (Instruction Cache, I$)
    - Most L2+ caches are "unified", can hold both instructions and data

- **Cache capacity:**
  - The total # of bytes in the cache

- **Cache line or cache block:**
  - A single entry in the cache

- **Cache block size:**
  - The number of bytes in each cache line
Even More Terms

- **Number of cache lines:**
  - Cache capacity / block size:

- **Cache associativity:**
  - The number of possible cache lines a given address may exist in.
  - Also the number of comparison operations needed to check for an element in the cache.
  - **Direct mapped:** A data element can only be in one possible location (N=1)
  - **N-way set associative:** A data element can be in one of N possible positions
  - **Fully associative:** A data element can be at any location in the cache.
    - Associativity == # of lines

- Total # of cache lines == capacity of cache/line size
- Total # of lines in a set == # of cache lines / associativity
Even More More terms: Parts of the address

- **Offset**: The lowest bits of the memory address which say where data exists within the cache line.
  - It is $\log_2(\text{line/block size})$
  - So for a cache with 64B blocks it is 6 bits

- **Index**: The portion of the address which says where in the cache an address may be stored.
  - Takes $\log_2(\# \text{ of cache lines} / \text{associativity})$ bits
  - So for a 4 way associative cache with 512 lines it is 7 bits

- **Tag**: The portion of the address which must be stored in the cache to check if a location matches.
  - # of bits of address - (# of bits for index + # of bits for offset)
  - So with 64b addresses it is 51b...
Even More More More \textit{More} terms...

\begin{itemize}
  \item \textbf{Eviction}:
    \begin{itemize}
      \item The process of removing an entry from the cache
    \end{itemize}
  \item \textbf{Write Back}:
    \begin{itemize}
      \item A cache which only writes data up the hierarchy when a cache line is evicted
      \item Instead set a \textit{dirty bit} on cache entries
        \begin{itemize}
          \item All i7 caches are \textit{write back}
        \end{itemize}
    \end{itemize}
  \item \textbf{Write Through}:
    \begin{itemize}
      \item A cache which always writes to memory
    \end{itemize}
  \item \textbf{Write Allocate}:
    \begin{itemize}
      \item If writing to memory \textit{not in the cache} fetch it first
        \begin{itemize}
          \item i7 L2 is Write Allocate
        \end{itemize}
    \end{itemize}
  \item \textbf{No Write Allocate}:
    \begin{itemize}
      \item Just write to memory without a fetch
        \begin{itemize}
          \item i7 L1 is no write allocate
        \end{itemize}
    \end{itemize}
\end{itemize}
Even Mostest Terms...

Cache Performance

- **Hit Time:**
  - Amount of time to return data in a given cache: depends on the cache
  - i7 L1 hit time: 4 clock cycles

- **Miss Penalty:**
  - Amount of *additional* time to return an element if its not in the cache: depends on the cache

- **Miss Rate:**
  - Fraction of a *particular program's* memory requests which miss in the cache

- **Average Memory Access Time (AMAT):**
  - Hit time + Miss Rate * Miss Penalty
Example: Direct-Mapped Cache with 4 Single-Word Blocks, Worst-Case Reference String

- Consider the main memory address reference string of word numbers: 0 4 0 4 0 4 0 4

Start with an empty cache - all blocks initially marked as not valid

- 8 requests, 8 misses

Ping-pong effect due to conflict misses - two memory locations that map into the same cache block
Alternative Block Placement Schemes

- **DM placement**: mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
- **SA placement**: four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
- **FA placement**: mem block 12 can appear in any cache blocks
Example: 4-Word 2-Way SA $ 
Same Reference String

- Consider the main memory address reference string
  0 4 0 4 0 4 0 4

Start with an empty cache - all blocks initially marked as not valid

<table>
<thead>
<tr>
<th></th>
<th>Mem(0)</th>
<th>Mem(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>010</td>
</tr>
<tr>
<td>4</td>
<td>000</td>
<td>010</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>010</td>
</tr>
<tr>
<td>4</td>
<td>000</td>
<td>010</td>
</tr>
</tbody>
</table>

- 8 requests, 2 misses
- Solves the ping-pong effect in a direct-mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!
Different Organizations of an Eight-Block Cache

Total size of $ in blocks is equal to \textit{number of sets} \times \textit{associativity}. For fixed $ size and fixed block size, increasing associativity decreases number of sets while increasing number of elements per set. With eight blocks, an 8-way set-associative $ is same as a fully associative $.
Range of Set-Associative Caches

- For a fixed-size cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit.

| Tag | Index | Word offset | Byte offset |
Range of Set-Associative Caches

- For a fixed-size cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit.

<table>
<thead>
<tr>
<th>Decreasing associativity</th>
<th>Increasing associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped (only one way)</td>
<td>Fully associative (only one set)</td>
</tr>
<tr>
<td>Smaller tags, only a single comparator</td>
<td>Tag is all the bits except block and byte offset</td>
</tr>
</tbody>
</table>

- For a **fixed-size** cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets — decreases the size of the index by 1 bit and increases the size of the tag by 1 bit.
Total Cache Capacity =

\[ \text{Associativity} \times \# \text{ of sets} \times \text{block}_\text{size} \]

\[ \text{Bytes} = \text{blocks/set} \times \text{sets} \times \text{Bytes/block} \]

\[ C = N \times S \times B \]

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Byte Offset</th>
</tr>
</thead>
</table>

\[ \text{address\_size} = \text{tag\_size} + \text{index\_size} + \text{offset\_size} \]

\[ = \text{tag\_size} + \log_2(S) + \log_2(B) \]
Clickers/Peer Instruction

• For a cache with constant total capacity, if we increase the number of ways by a factor of 2, which statement is false:
  • A: The number of sets could be doubled
  • B: The tag width could decrease
  • C: The block size could stay the same
  • D: The block size could be halved
  • E: Tag width must increase
Costs of Set-Associative Caches

• N-way set-associative cache costs
  • N comparators (delay and area)
  • MUX delay (set selection) before data is available
  • Data available after set selection (and Hit/Miss decision). DM $: block is available before the Hit/Miss decision
    • In Set-Associative, not possible to just assume a hit and continue and recover later if it was a miss

• When miss occurs, which way’s block selected for replacement?
  • *Least Recently Used* (LRU): one that has been unused the longest (principle of temporal locality)
    • Must track when each way’s block was used relative to other blocks in the set
    • For 2-way SA $, one bit per set → set to 1 when a block is referenced; reset the other way’s bit (i.e., “last used”)
Cache Replacement Policies

- Random Replacement
  - Hardware randomly selects a cache evict

- Least-Recently Used
  - Hardware keeps track of access history
  - Replace the entry that has not been used for the longest time
  - For 2-way set-associative cache, need one bit for LRU replacement

- Example of a Simple “Pseudo” LRU Implementation
  - Assume 64 Fully Associative entries
  - Hardware replacement pointer points to one cache entry
  - Whenever access is made to the entry the pointer points to:
    - Move the pointer to the next entry
    - Otherwise: do not move the pointer
  - (example of “not-most-recently used” replacement policy)
Benefits of Set-Associative Caches

- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)
Administrivia...

- New extended due dates for Project 3A and 3B
  - 3A -> 3/20 (Friday)
  - 3B -> 4/3 (Friday)
- Project party tomorrow
  - Details TBD
  - Probably another project party after Spring Break
- Midterm 2 canceled.
- Going to try to set up a standing "CS61C room" in Zoom where you can just connect and chat with everyone in the class about, well, whatever
Sources of Cache Misses (3 C’s)

- **Compulsory** (cold start, first reference):
  - 1st access to a block, not a lot you can do about it.
  - If running billions of instructions, compulsory misses are insignificant

- **Capacity**:
  - Cache cannot contain all blocks accessed by the program
  - Misses that would not occur *with infinite cache*

- **Conflict** (collision):
  - Multiple memory locations mapped to same cache set
  - Misses that would not occur with ideal *fully associative cache of the same size*
Prefetching...

- **Programmer/Compiler:** I know that, later on, I will need this data...🤔
- **Tell the computer to** *prefetch* **the data**
  - Can be as an explicit prefetch instruction
  - Or an implicit instruction:
    \[ \text{lw 0 0(t0)} \]
    - Won't stall the pipeline on a cache miss: The processor control logic recognizes this situation
- **Allows you to hide the cost of compulsory misses**
  - You still need to fetch the data however
Improving Cache Performance

AMAT = Time for a hit + Miss rate \times Miss penalty

- Note: miss penalty is the \textit{additional} time required for a cache miss
- Reduce the time to hit in the cache
  - E.g., Smaller cache
- Reduce the miss rate
  - E.g., Bigger cache
    Long cache lines (somewhat: improves ability to exploit spacial locality at the cost of reducing the ability to exploit temporal locality)
  - E.g., Better programs!
- Reduce the miss penalty
  - E.g., Use multiple cache levels
Cache Design Space

Computer architects expend considerable effort optimizing organization of cache hierarchy – big impact on performance and power!

- Several interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Replacement policy
  - Write-through vs. write-back
  - Write allocation
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost
- Simplicity often wins
Primary Cache Parameters

- **Block size**
  - how many bytes of data in each cache entry?

- **Associativity**
  - how many ways in each set?
  - Direct-mapped => Associativity = 1
  - Set-associative => 1 < Associativity < #Entries
  - Fully associative => Associativity = #Entries

- **Capacity (bytes) = Total #Entries * Block size**

- **#Entries = #Sets * Associativity**
Clickers

• For fixed capacity and fixed block size, how does increasing associativity effect AMAT?

A: Increases hit time, decreases miss rate
B: Decreases hit time, decreases miss rate
C: Increases hit time, increases miss rate
D: Decreases hit time, increases miss rate
Administrivia

• Project 3.2:
  • Still due on March 23rd...
  • But the rest of "Spring Break" counts as only a single day for lateness!
• HW over spring break has been extended as well
• Project Partays!!!!
• Midterm Survey, please fill it out!
Increasing Associativity?

- **Hit time as associativity increases?**
  - Increases, with large step from direct-mapped to \( \geq 2 \) ways, as now need to mux correct way to processor
  - Smaller increases in hit time for further increases in associativity
    - Able to build reasonably efficient wide muxes

- **Miss rate as associativity increases?**
  - Goes down due to reduced conflict misses, but most gain is from \( 1 \rightarrow 2 \rightarrow 4 \)-way with limited benefit from higher associativities

- **Miss penalty as associativity increases?**
  - Mostly unchanged, replacement policy runs in parallel with fetching missing line from memory
Increasing #Entries?

- Hit time as #entries increases?
  - Increases, since reading tags and data from larger memory structures

- Miss rate as #entries increases?
  - Goes down due to reduced capacity and conflict misses
  - Architects rule of thumb: miss rate drops ~2x for every ~4x increase in capacity (only a gross approximation)

- Miss penalty as #entries increases?
  - Unchanged

  At some point, increase in hit time for a larger cache may overcome the improvement in hit rate, yielding a decrease in performance
Clickers: Impact of larger blocks on AMAT

- For fixed total cache capacity and associativity, what is effect of larger **blocks** on each component of AMAT:
  - A: Decrease, B: Unchanged, C: Increase, D: \(\_\(\_\(\_\_\(\_\(\_\(\_\_\_\)\_\_\_\)\_\_\_\)\_\_\_\)\_\_\_\)\_\_\_\)
- Hit Time?
- Miss Penalty?
- Miss Rate?
Increasing Block Size?

- Hit time as block size increases?
  - Hit time unchanged, but might be slight hit-time reduction as number of tags is reduced, so faster to access memory holding tags

- Miss rate as block size increases?
  - Goes down at first due to spatial locality, then increases due to increased conflict misses due to fewer blocks in cache

- Miss penalty as block size increases?
  - Rises with longer block size, but with fixed constant initial latency that is amortized over whole block
How to Reduce Miss Penalty?

- Could there be locality on misses from a cache?
- Use multiple cache levels!
- With Moore’s Law, more room on die for bigger L1 caches and for second-level (L2) cache
- And now big L3 caches!
- IBM mainframes have ~1GB L4 cache off-chip.
Review: Memory Hierarchy

As we move to outer levels the latency goes up and price per bit goes down.

Increasing distance from processor, decreasing speed.

Levels in memory hierarchy:
- **Inner**
- **Outer**

Size of memory at each level:
- Level 1
- Level 2
- Level 3
- ... 
- Level n
24% of CPU access miss in L1

15% also miss in L2

4% also miss in L3

L1 Cache: 32KB I$, 32KB D$
L2 Cache: 256 KB
L3 Cache: 4 MB

FIGURE 5.47 The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPECCPU2006 benchmarks.
Local vs. Global Miss Rates

- **Global** miss rate – the fraction of references that miss some level of a multilevel cache
  - misses in this cache divided by the total number of memory accesses generated by the CPU
- **Local** miss rate – the fraction of references to one level of a cache that miss
  - Local Miss rate L2$ = \frac{L2$ Misses}{L1$ Misses} = \frac{L2$ Misses}{total\_L2\_accesses}
  - L2$ local miss rate >> than the global miss rate
Clickers/Peer Instruction:
Graph is for **global** miss rate

- Overall, what are L2 and L3 **local** miss rates?

A: L2 > 50%, L3 > 50%
B: L2 ~ 50%, L3 < 50%
C: L2 ~ 50%, L3 ~ 50%
D: L2 < 50%, L3 < 50%
E: L2 > 50%, L3 ~50%
Local vs. Global Miss Rates

- **Local miss rate** – the fraction of references to one level of a cache that miss
  - Local Miss rate $L2$ = $L2$ Misses / $L1$ Misses

- **Global miss rate** – the fraction of references that miss in all levels of a multilevel cache
  - $L2$ local miss rate $>>$ than the global miss rate
  - Global Miss rate = $L2$ Misses / Total Accesses
    = $(L2$ Misses / $L1$ Misses) $\times$ $(L1$ Misses / Total Accesses)
    = Local Miss rate $L2$ $\times$ Local Miss rate $L1$

- **AMAT** = Time for a hit + Miss rate $\times$ Miss penalty
  - For 2-level cache system:
    AMAT = Time for a $L1$ hit + Miss rate $L1$ $\times$
    (Time for a $L2$ hit + (local) Miss rate $L2$ $\times$ $L2$ Miss penalty)
# Real World Caches: Nehalem and Barcelona

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4 (Barcelona)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 cache organization</strong></td>
<td>Split instruction and data caches</td>
<td>Split instruction and data caches</td>
</tr>
<tr>
<td><strong>L1 cache size</strong></td>
<td>32 KB each for instructions/data per core</td>
<td>64 KB each for instructions/data per core</td>
</tr>
<tr>
<td><strong>L1 block size</strong></td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L1 write policy</strong></td>
<td>Write-back, Write-allocate</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td><strong>L1 hit time (load-use)</strong></td>
<td>Not Available</td>
<td>3 clock cycles</td>
</tr>
<tr>
<td><strong>L2 cache organization</strong></td>
<td>Unified (instruction and data) per core</td>
<td>Unified (instruction and data) per core</td>
</tr>
<tr>
<td><strong>L2 cache size</strong></td>
<td>256 KB (0.25 MB)</td>
<td>512 KB (0.5 MB)</td>
</tr>
<tr>
<td><strong>L2 block size</strong></td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L2 write policy</strong></td>
<td>Write-back, Write-allocate</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td><strong>L2 hit time</strong></td>
<td>Not Available</td>
<td>9 clock cycles</td>
</tr>
<tr>
<td><strong>L3 cache organization</strong></td>
<td>Unified (instruction and data)</td>
<td>Unified (instruction and data)</td>
</tr>
<tr>
<td><strong>L3 cache size</strong></td>
<td>8192 KB (8 MB), shared</td>
<td>2048 KB (2 MB), shared</td>
</tr>
<tr>
<td><strong>L3 block size</strong></td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L3 write policy</strong></td>
<td>Write-back, Write-allocate</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td><strong>L3 hit time</strong></td>
<td>Not Available</td>
<td>38 (?)clock cycles</td>
</tr>
</tbody>
</table>
A Modern x86's Organization: Intel Xeon E7 v3 (Haswell EX)

- A 18 core processor!
  - And you can get more cores today... The v4 comes in a 24 core version!?!?
- Each core can run two separate threads
  - Two *separate* program counters
  - Very pipelined: 14-19 stages (depending on actual instruction)
  - Very superscalar: Issuing up to 7 \(\mu\)OP per cycle between the two threads
  - Very out-of-order: 168 actual registers, 192 instruction window for reordering
- Addressing:
  - 64b addressing, 64B block size for all caches
The Caches

• Each core, 32kB 4-way associative L1 instruction cache, 64B block size
  • 4 cycles latency, pipelined
    • So the first 4 stages of the instruction pipeline!
  • 512 cache lines
  • Offset: lg(64) = 6b, Index: lg((512/4)) = 7b, Tag: 64-13 = 51b

• Each core, 32 kB L1, 8-way associative write-back data cache
  • 4 cycles latency, pipelined, write back but no write allocate!
  • Pseudo-LRU replacement

• Each core, 256kB 8-way associative write-back L2 cache
  • 10 cycles latency, write back with write allocate
  • Pseudo-LRU

• Common cache, 45MB 16-way associative unified L3 cache (2.5MB per core)
  • Each core has its own section of cache, but all cores can read/write all entries
  • Almost-random replacement
Die Photograph:
18 core Intel Haswell EX processor

- Can see the patterns:
  - One Core Block
  - One L3 Cache Block
  - I/O Control
# CPI/Miss Rates/DRAM Access SpecInt2006

<table>
<thead>
<tr>
<th>Name</th>
<th>CPI</th>
<th>L1 D cache misses/1000 instr</th>
<th>L2 D cache misses/1000 instr</th>
<th>DRAM accesses/1000 instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>0.75</td>
<td>3.5</td>
<td>1.1</td>
<td>1.3</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.85</td>
<td>11.0</td>
<td>5.8</td>
<td>2.5</td>
</tr>
<tr>
<td>gcc</td>
<td>1.72</td>
<td>24.3</td>
<td>13.4</td>
<td>14.8</td>
</tr>
<tr>
<td>mcf</td>
<td>10.00</td>
<td>106.8</td>
<td>88.0</td>
<td>88.5</td>
</tr>
<tr>
<td>go</td>
<td>1.09</td>
<td>4.5</td>
<td>1.4</td>
<td>1.7</td>
</tr>
<tr>
<td>hmmer</td>
<td>0.80</td>
<td>4.4</td>
<td>2.5</td>
<td>0.6</td>
</tr>
<tr>
<td>sjeng</td>
<td>0.96</td>
<td>1.9</td>
<td>0.6</td>
<td>0.8</td>
</tr>
<tr>
<td>libquantum</td>
<td>1.61</td>
<td>33.0</td>
<td>33.1</td>
<td>47.7</td>
</tr>
<tr>
<td>h264avc</td>
<td>0.80</td>
<td>8.8</td>
<td>1.6</td>
<td>0.2</td>
</tr>
<tr>
<td>omnetpp</td>
<td>2.94</td>
<td>30.9</td>
<td>27.7</td>
<td>29.8</td>
</tr>
<tr>
<td>astar</td>
<td>1.79</td>
<td>16.3</td>
<td>9.2</td>
<td>8.2</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>2.70</td>
<td>38.0</td>
<td>15.8</td>
<td>11.4</td>
</tr>
<tr>
<td>Median</td>
<td>1.35</td>
<td>13.6</td>
<td>7.5</td>
<td>5.4</td>
</tr>
</tbody>
</table>
In Conclusion, Cache Design Space

- Several interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Replacement policy
  - Write-through vs. write-back
  - Write-allocation

- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost

- Simplicity often wins
More Misses…

- We have **Compulsory**, **Capacity**, and **Conflict**…
- We also have **Coherence**
  - Two different processor may share memory...
    - They implement cache coherence so that both processors see the same shared memory
    - When one processor writes to memory, it invalidates the other processor's cache entry for that memory
  - Thus if both processors are working on the same data...
    - This causes Coherence misses
- A related problem can occur if one shared cache is working on two unrelated problems
  - You get additional **capacity** misses:
    Can happen in "multithreaded" (aka 'Intel Hyperthreaded') processor cores
Fun Additional Stuff: Nick's Caches

- Note: These won't be on the exam, but they are interesting asides
  - Nick's research has used this material in multiple ways
- Predictability and caches
  - Why it's bad
  - Unpredictable caches: Permutation caches and location-associative permutation caches
Predictability and Caches

• Caches improve performance but…
  • The performance improvement depends on the input
    • E.g. conflict misses depend on input patterns

• An attacker can take advantage of this
  • Timing of operations can tell something about the input
    • E.g. techniques to extract cryptographic keys!
  • Attacker selected inputs can degrade performance
Why Timing Matters

- Timing enables "side-channel" attacks on cryptography
  - The ability to know some detail of an encryption system based on how long operations take
    - Part of a larger class of side-channel attacks
- It is a fundamentally hard problem to build cryptographic systems that don't have sidechannels
  - Modern processors make this even harder
- Take 161 and 261 for more on the deep voodoo that side channels can do

From "CacheBleed: A Timing Attack on OpenSSL Constant-Time RSA"
Attacker Selected Input

- Alternatively, if the attacker can select the input...
  - The attacker can select **hard** input: E.G. Traffic that causes ping-ponging in a direct mapped cache

- Nick's problem:
  - He had to cache IP addresses (32 bit values)
    - This is a network application for security
  - He only wants to store a small amount of information
    - On chip storage expensive (in this case, on an FPGA)
  - He had plenty of room to pipeline
    - Each network packet is independent: No need to forward
  - Misses are expensive
    - Requires processing the packet in software
Nick's Cache #1: Permutation Cache

- Traditionally, you would hash the address
  - With a "salt" to randomize things
    - Attacker needs to break the hash function to predict whether two different addresses will match to the same location
  - But this requires storing the whole original IP for your tag
- So instead of a hash, use a 32b *keyed permutation*
  - Aka a 32b block cypher
- Now you can use a conventional tag/index approach
  - Requires only storing the tag -> space mattered in this application
Nick's Cache #2: Location Associativity

- The fabric Nick used allowed "dual-ported" memories
  - Like your register file on your processor design: two independent read ports that operate at the same time

- Rather than using set associativity…
  - Instead do two different permutations (keys) and have one of two possible locations

- If X, Y, and Z map to the same location with one key...
  - They probably do not on the other key: fewer conflict misses
  - Even better, can probably move a value to further reduce conflict misses
Simulation… **Actual** Occupancy for Caching Random Elements
And Since the Permutation looks "random"...

- It is one of those cases where simulation matches reality...
  - Because I'm caching *random* values
- Most cache studies are much more program dependent
- But it does give a way of reducing *conflict* misses without having to increase associativity
  - Just need to calculate two permutations in parallel rather than just one