Even More Caches: Successes & Failures

Outline...

- A Simple Program to Analyze Caches
- Branch Prediction
- Multiprocessors and Caches
  - And what that means
- Virtual Memory is a Cache
  - And what that means

So Lets Analyze Caches...

- A conceptually simple set of nested for loop:

  -
    ```
    int array[MAXLEN]
    for(size = 1024; size <= MAXLEN; size = size << 1){
      for(stride = 1; stride <= size; stride = stride << 1){
        // Some initialization to eliminate compulsory misses
        // Repeat this loop enough to get good timing for computing AMAT
        for(i = 0; i < size; i += stride){
          array[i] = array[i] + i
        }
        // Now add some timing information for how long the
        // for loop took
    }}
    ```
- This is ***striding access***:
  - Rather than every element in the array this accesses every ***i*th** element
- This is ***designed to break caches***:
  - Caches tend to work great up until the instant they don't...
  - So by seeing where and how the cache falls down, this can deduce the internal structure

Reminder: Cache Miss Types

- ***Compulsory***: A miss that occurs on first load
  - An infinitely large cache would still incur the miss
- ***Capacity***: A miss that occurs because the cache isn't big enough
  - An infinitely large cache would not miss
- ***Conflict***: A miss that occurs because the associativity doesn't allow the items to be stored
  - A fully associative cache of the same size would not miss

Reminder: Other Parameters

- Block or Line size:
  - The # of bytes in each entry
- Associativity:
  - The degree of flexibility in storing a particular address
    - Direct mapped: One location
    - N-way set associative: one of N possible locations
    - Fully associative: Any location
- AMAT: Average Memory Access Time
  - hit time + miss penalty * miss rate

Breaking Caches: Capacity...

- Up until the test exceeds the cache capacity...
  - Everything is fine!
- But once sizeof(array) > cache size...
  - Then things break down and you start getting misses
- Which increases the loop time
  - AMAT = hit time + miss penalty * miss rate
- So where the size breaks capacity...

Breaking Caches: Spacial Locality

- Spacial locality breaks down if only a ***single*** entry in each cache line is ever accessed
  - Since the rest of the cache line provides no benefit...
- So worst-case behavior occurs when each line is only accessed in one location
  - So when stride * sizeof(int) == block size
- Combined with where the capacity break occurs...
  - And you now know the line-size

Un-breaking Caches: Associativity

- If your array is 2x the cache capacity...
  - But you are striding at >= 2*block size...
- You aren't using all the cache entries
  - So by definition, all your misses are no longer capacity misses but ***conflict*** misses!
- Reminder: Tag/Index/Offset...
  - The index specifies the possible location for a set associative cache
  - So when do the accesses have different indexes?
    - That is when you have stopped having conflict misses

Breaking Caches: Levels...

- Each level is its own cache...
  - To test L2, you must be using references that break L1...
- Which is fine for capacity, but...
  - If L2 line size <= L1 line size, we can't reliably tell
    - Since the L1 cache provides the spacial locality
    - But generally most multi-level caches use the same linesize, defined by the external memory interface
  - If the L2 associativity <= L1 associativity
    - The conflict misses will be removed in L1

Actual Test: Raspberry Pi: L1 Cache hitting...

```
• Size (bytes):   32768 Stride (bytes):     4 read+write:     9 ns
• Size (bytes):   32768 Stride (bytes):     8 read+write:     7 ns
• Size (bytes):   32768 Stride (bytes):    16 read+write:     8 ns
• Size (bytes):   32768 Stride (bytes):    32 read+write:     9 ns
• Size (bytes):   32768 Stride (bytes):    64 read+write:     9 ns
• Size (bytes):   32768 Stride (bytes):   128 read+write:     7 ns
• Size (bytes):   32768 Stride (bytes):   256 read+write:     9 ns
• Size (bytes):   32768 Stride (bytes):   512 read+write:     9 ns
• Size (bytes):   32768 Stride (bytes):  1024 read+write:     8 ns
• Size (bytes):   32768 Stride (bytes):  2048 read+write:     9 ns
• Size (bytes):   32768 Stride (bytes):  4096 read+write:     8 ns
• Size (bytes):   32768 Stride (bytes):  8192 read+write:     8 ns
• Size (bytes):   32768 Stride (bytes): 16384 read+write:     8 ns
• Size (bytes):   32768 Stride (bytes): 32768 read+write:     9 ns
```

Actual Test: Raspberry Pi: L1 missing...

```
• Size (bytes):   65536 Stride (bytes):     4 read+write:    7 ns
• Size (bytes):   65536 Stride (bytes):     8 read+write:    8 ns
• Size (bytes):   65536 Stride (bytes):    16 read+write:    9 ns
• Size (bytes):   65536 Stride (bytes):    32 read+write:    8 ns
• Size (bytes):   65536 Stride (bytes):    64 read+write:    8 ns
• Size (bytes):   65536 Stride (bytes):   128 read+write:   10 ns
• Size (bytes):   65536 Stride (bytes):   256 read+write:   10 ns
• Size (bytes):   65536 Stride (bytes):   512 read+write:   14 ns
• Size (bytes):   65536 Stride (bytes):  1024 read+write:   16 ns
• Size (bytes):   65536 Stride (bytes):  2048 read+write:   18 ns
• Size (bytes):   65536 Stride (bytes):  4096 read+write:   20 ns
• Size (bytes):   65536 Stride (bytes):  8192 read+write:   18 ns
• Size (bytes):   65536 Stride (bytes): 16384 read+write:    8 ns
• Size (bytes):   65536 Stride (bytes): 32768 read+write:    8 ns
• Size (bytes):   65536 Stride (bytes): 65536 read+write:    9 ns
```

Actual Testing: Watching L2 Fail

```
• Size (bytes): 1048576 Stride (bytes):    4 read+write:   8 ns
• Size (bytes): 1048576 Stride (bytes):    8 read+write:   9 ns
• Size (bytes): 1048576 Stride (bytes):   16 read+write:  12 ns
• Size (bytes): 1048576 Stride (bytes):   32 read+write:  29 ns
• Size (bytes): 1048576 Stride (bytes):   64 read+write:  61 ns
• Size (bytes): 1048576 Stride (bytes):  128 read+write:  61 ns
• Size (bytes): 1048576 Stride (bytes):  256 read+write:  61 ns
• Size (bytes): 1048576 Stride (bytes):  512 read+write: 117 ns
• Size (bytes): 1048576 Stride (bytes): 1024 read+write: 117 ns
• Size (bytes): 1048576 Stride (bytes): 2048 read+write: 124 ns
• Size (bytes): 1048576 Stride (bytes): 4096 read+write: 140 ns
• Size (bytes): 1048576 Stride (bytes): 8192 read+write:  61 ns
• Size (bytes): 1048576 Stride (bytes): 16384 read+write:  24 ns
• Size (bytes): 1048576 Stride (bytes): 32768 read+write:  24 ns
• Size (bytes): 1048576 Stride (bytes): 65536 read+write:  21 ns
• Size (bytes): 1048576 Stride (bytes): 131072 read+write:  18 ns
• Size (bytes): 1048576 Stride (bytes): 262144 read+write:   8 ns
• Size (bytes): 1048576 Stride (bytes): 524288 read+write:   7 ns
• Size (bytes): 1048576 Stride (bytes): 1048576 read+write:   8 ns
```

Can Already See How Caches Fail Catastrophically...

- For a memory-bound task...
  - Fit in L1 cache: AMAT 7ns
  - Fit in L2 cache: AMAT ~20ns
  - Exceed L2 cache: AMAT 100+ns
- Performance drops by an ***order of magnitude*** when you exceed the capabilities of the cache even by not that much!

Complications...

- Of course, why we don't inflict this particular assignment on you anymore...
  - There are a lot of additional complications on modern processors
- Memory fetching/prefetching...
  - L2 cache is starting to hit memory at a stride of 64, but...
  - Performance keeps getting worse until the stride is larger
    - Memory is probably transferring 256B at a time to L1 as well as L2...
- There may be a "victim cache"
  - A small fully associative cache that holds the last few evicted cache lines:
    So although L2 is only 16 way according to the documentation I find, we still are getting good performance on the 32 way and 64 way test
- And this is on a ***simple*** modern processor
  - ***Only*** 4 cores, ***only*** 2-issue superscalar
- This ***used*** to be a good homework assignment

More on Victim Caches...

- Observation: Conflict misses are a pain, but...
  - Perhaps a little associativity can help without having to be a fully associative cache
- In addition to the main cache...
  - Have a very small (16-64 entry) ***fully associative*** "victim" cache
- Whenever we evict a cache entry
  - Don't just get rid of it, put it in the victim cache
- Now on cache misses...
  - Check the victim cache first, if it is in the victim cache you can just reload it from there

Another Pathological Example...

-
  ```
  int j, k, array[256*256];
  for (k = 0; k < 255; k++){
    for (j = 0; j < 256; j++){
        array[256*j] += array[256*j + k + 1]
  }}
  ```
- This from the homework has a nasty pathology...
  - It experiences ***no*** spacial locality of note: both array reads and the array write are a stride of 256 entries
  - And in the example for the homework it also generates a huge number of capacity misses

But a minor tweak...

-
  ```
  int j, k, array[256*256];
  for (j = 0; j < 256; j++){
    for (k = 0; k < 255; k++){
        array[256*j] += array[256*j + k + 1]
  }}
  ```
- And now it runs vastly better as it changes from stride 256 to stride 1 and stride 0...
  - Stride 0 == best case temporal locality
  - Stride 1 == best case spacial locality

Clicker Question...

- Adding an additional layer of cache hierarchy (e.g. L4 in a system with L1-L3) will:
  A) Improve. B) Not affect. C) Make Worse. D) 🤷‍♂️
  the following properties:
  - Lower level cache hit time
  - Lower level cache miss rate
  - Lower level cache miss penalty
  - AMAT on all workloads
  - (hopefully) AMAT on most workloads

Administrivia

- Project Party: Wednesday, 7-10pm in 293 Cory
- Guerrilla session on Caches Thursday from 7-9pm, Barrows 20
- Midterm 2 is Tuesday, 3/20 from 8-10pm.
  - Everyone (including DSP and conflict) should have already heard from Dylan or me if you have alternative exam arrangements
  - Reminder, you are allowed ***two*** hand-written, double sided note sheets
- Midterm Review this Sunday March 18th 1-4PM GPBB 100
- HW3 the cache part due Friday 3/16

Blocking-out Data

- Very common motif (yes, I like golang...):

  -
    ```
    for i := range A {
       for j := range B {
          fn(A[i],B[j])
    }}
    ```
  - "Do something for every pair of elements"
- If B fits in the cache...
  - Its all good...
- But if B doesn't...
  - The inner iteration is going to be dominated by ***capacity*** misses as B has to keep being reloaded
    - So there is no more temporal locality for B[j]
  - But the fetches of A are still going to be fine because a lot of ***temporal locality*** for A[i]
- And its going to be very good ***up until the instance it isn't***
  - Caches don't tend to degrade performance gracefully... Instead you get step-functions

Implications...

- If one array is a lot bigger than the other...
  - It should be the outer one in the loop: It doesn't generally matter if the outer one doesn't fit
    - Since there is tons of temporal locality for the outer array that the cache will take advantage of
- But if both don't fit, you need to be better

  -
    ```
    for i := 0; i < len(A); i += BLOCK {
       for j := 0; j < len(B); j += 1 {
          for k :=0; k < BLOCK; k++ {
              if (k + i) < len(A) {
                  fn(A[i*block+k],B[j])
    }}}}
    ```
  - Now have a lot more temporal locality for the entries of both A and B, ***if BLOCK is set correctly***

Another Cache: Branch Predictor

- In our simple pipeline, we assume branches-not-taken
  - Always start fetching the next instruction
- If a branch or jump is taken...
  - Then we have to kill the non-taken instructions so they don't cause side effects
- But both branches and jumps are PC relative...
  - So if we can quickly look at the instruction and decide 'eh, probably taken/not' we can compute the new location for the PC if we can guess right
    - Which for `jal` we always can, but branches we need to guess
- Idea: ***branches*** have temporal locality!
  - Loops: `for (x = 0; x < n...)`
  - Rare conditionals: `if (err) ...`

A Simple Branch Predictor

- Have an N entry, direct-mapped memory
  - EG, a 1024x1b memory
- If fetched instruction is a branch...
  - Check if the bit for pc[12:2] is set...
    - If so, set next PC to PC + branch offset for fetching
    - Set bit in pipeline to say "branch predicted-taken"
- When actually evaluating branch in EX...
  - Set pc[12:2] to branch taken/not-taken status
  - If branch taken but predicted not-taken
    - ***Kill untaken instructions***
  - If branch not taken but predicted taken
    - ***Kill predicted instructions***

Improving it slightly...

- How about 2 bits:
  - Each entry starts at 01...
  - If taken, increment with saturating arithmetic (so max is 11)
  - If not taken, decrement by one (so min is 00)
- If the upper bit is 1, assume the branch is taken
  - Now a function with a commonly taken loop will only mispredict once rather than twice
- Miss penalty for a mispredicted branch: The # of instructions that got terminated because of the wrong prediction
  - EG, on a dual-issue, 10-deep 2x superscalar like a Raspberry Pi or smartphone: Probably ~10 instructions
  - On a modern x86? It could be 20+
- Can then try even fancier predictors...
  - But we get into ***diminishing returns:***
    - The simple-smart thing (e.g. a two bit branch predictor) is a big win...
    - But trying to get fancier no longer helps nearly as much

Approximate Cache...

- The data caches are exact:
  - They will return an answer that is exactly what is asked for
- But this branch predictor is approximate:
  - It can make mistakes due to aliasing:
    Its not actually storing the full address as a tag to check
- Sometimes its OK to be wrong
  - So data structures that are this way are also particularly interesting...
  - EG "Bloom Filters": One of the ***coolest*** data structures on the planet.
    - Is this element in the set? Using a small amount of memory
      - Yes it is? Will always return "Yes"
      - No it isn't? Will mostly return "No", but can have an error

A related cache: return target location...

- Observation:
  - On RISC-V, you call a function with **`jal`** or **`jalr`** (object oriented) with the return set to **`ra`**
  - And you return with a **`jalr`** with the source register as **`ra`**
- So lets maintain a small stack in hardware...
  - Whenever we see **`jal`** or **`jalr`** writing to **`ra`**:
    We write PC+4 into the stack
  - Whenever we see **`jalr`** reading from **`ra`**:
    We predict the top of this stack as the next PC, and pop this stack
- Result: We should always correctly predict a function return address...
  - Works as long as we don't exceed the stack depth: once we hit that we will start getting misses

And A Final Related Cache: Branch Target Buffer

- Function calls using **`jal`** we will never mispredict on RISC-V
  - Since they are all PC relative
- But so much today is object-oriented programming which uses **`jalr`**:
  C++ and Java object calls are equivalent to calling pointers to functions
  - **`foo.bar()`** is implemented as something like this:

```
lw t0 0(a0) # Get the pointer to the "virtual function table" in the object
lw t0 8(t0) # Get the pointer to the function to actually call
jalr ra t0  # Do a JALR to call bar(),
            # with the object foo as the first implicit argument
```

- So cache this as well:
  - On a **`jalr`** which writes to **`ra`**...
    Look in a small cache for the address to predict to based on current PC
  - When evaluating the jump, set the value in this cache to the address used
- It is the x86 equivalent of this cache that is part of one of the Spectre vulnerabilities

Caches and Multiprocessors...

- These days practically every computer is a multiprocessor
  - Since that is the only way we know how to increase computation by throwing more silicon at the problem
  - But we can't make single-processor performance ***worse*** in this process
    - So these processors ***must*** have significant caches
  - And because the L1 caches are integrated into the pipeline, to prevent structural hazards each processor must have its ***own*** caches
- What happens if multiple processors are accessing the same piece of memory?

Multiple Processor Reading Memory?

- No problem!
  - Each processor just caches the data independently
- There is ***no*** issue with multiple processors reading the same thing
  - Their own caches have a unique copy...
    But the values should always be the same

Multiple Processors Writing?

- We need ***coherency***: Writes from one processor ***must*** be reflected in memory that other processors read after some short period of time
  - There have been processors made without this, but it is impossible to program these
- Goal is to guarantee the following property:
  - If processor A writes to memory location L, ***within time T***, all other processors will see the updated data

Idea: Broadcasting Messages...

- We need a way for the processors to communicate
  - So we have some sort of fabric
    - It could be a shared bus
    - It could be something looking more like a packet-switched computer network
- Each processor (or more precisely its cache) can send and receive messages
  - Requests are "broadcast", a single sender can send a message to anybody...
  - Replies may or may not be broadcast: Can go to everyone or could go to a specific recipient

Broadcasting Writes...

- Processor ***A*** wants to write to physical location ***L*** for the first time...
  - First do a write-allocate...
  - It then sets the ***dirty*** bit on the cache
  - And ***broadcasts*** a message that "I am writing to ***L***"
- All other processors which receive the message
  - Do I have address ***L*** cached?
  - If no: Do nothing
  - If yes: ***invalidate*** the entry in the cache
    - ***Snooping*** on requests:
      Term comes from when all processors shared the same memory bus to communicate with the memory

Broadcasting Reads

- If there is a miss in the upper level of the cache in the processor...
  - Broadcast a read request: "Hey, does anyone have location ***L***?"
- If nobody has written to this location...
  - The memory controller/common cache just does a fetch and returns it: Just like any other cache miss
- If a processor has this location with the dirty bit set...
  - It goes "Hey, I have this"
  - It ***flushes*** the entry (writing the value to memory) and clears the dirty bit
  - It then says the new value to the requesting processor

Why Does This Work?

- If processor A wants to write to a non-dirty line...
  - If the element is in the cache...
    - It will write, and all other processors ***invalidate:*** If they then want to read that location they will have to broadcast that request
  - If not, it performs a read first...
    - So if reads are correct, it is going to be correct from there
- If processor B wants to read...
  - If the element is in its cache...
    - It is correct, because if someone else wrote to that location ***it would be invalid already***
  - If the element is not in its cache...
    - It will get the correct value from either another processor or the right location
    - And that other processor will now know it can't write because the dirty bit got cleared

CPU-0 reads byte at `0xdeadbeef`...

Upper Level Cache & Memory...

CPU-0: I want to read 0xdeadbee0

Cache Controller: value of 0xdeadbee0 is 0xf00dd00dca11c003

CPU-0

Address	Data	V	D
		0	0
		0	0
		0	0
		0	0

CPU-1

Address	Data	V	D
		0	0
		0	0
		0	0
		0	0

CPU-1 reads byte at `0xdeadbeef`...

Upper Level Cache & Memory...

CPU-1: I want to read 0xdeadbee0

Cache Controller: value of 0xdeadbee0 is 0xf00dd00dca11c003

CPU-0

Address	Data	V	D
deadbee0	**f00dd00dca11c003**	1	0
		0	0
		0	0
		0	0

CPU-1

Address	Data	V	D
deadbee0	**f00dd00dca11c003**	1	0
		0	0
		0	0
		0	0

CPU-1 writes data at **`0xdeadbee0`**...

Upper Level Cache & Memory...

CPU-1: I want to start writing to 0xdeadbee0

CPU-0

Address	Data	V	D
deadbee0	**f00dd00dca11c003**	0	0
		0	0
		0	0
		0	0

CPU-1

Address	Data	V	D
deadbee0	**cafef00dda016666**	1	1
		0	0
		0	0
		0	0

CPU-0 reads byte at `0xdeadbeef`...

Upper Level Cache & Memory...

CPU-0: I want to read 0xdeadbee0
CPU-1: value of 0xdeadbee0 is 0xcafef00dda016666

CPU-0

Address	Data	V	D
deadbee0	**cafef00dda016666**	1	0
		0	0
		0	0
		0	0

CPU-1

Address	Data	V	D
deadbee0	**cafef00dda016666**	1	0
		0	0
		0	0
		0	0

So Enter a New Miss Type: ***Coherence***

- A coherence miss occurs when two processes want to access the same data
  - Coherence misses are caused only by ***writes***, not ***reads***
    - The write will invalidate all ***other*** caches
- It means there is an ***anti-pattern*** that can cause performance artifacts on multiprocessors
  - Multiple processes reading to the same memory? Sure!
  - But if one starts writing to that memory...
    - The ***other*** processor will start getting coherency misses
    - But such misses also only go up to the shared cache:
      Why multiprocessors, when possible, use a shared cache between all processors
  - Reasonably easy to avoid ***with proper program structure***

Oh, and *Incoherence misses*

- What about when we have multiple processes running on the same processor
  - A modern x86 creates two "virtual" processors which share resources ("Hyperthreading")
- If those two processes have the same working set...
  - Great!
- If those two processes have different working sets...
  - This effectively acts like reducing the cache size with the corresponding increase in the miss rate

Virtual Memory Paging As A Cache...

- How virtual memory works we will cover later...
- But for now, its easy to model as a basic cache...
- Your program is given the illusion of as much RAM as it wants...
  - But only on the condition that it actually doesn't want it! :)
- Idea: Virtual memory can copy "pages" between RAM and disk
  - The main memory thus acts as a cache for the disk...

But What Are The Properties...

- Associativity: Effectively fully associative
- Replacement policy: Effectively full LRU
- Block size: 4kB or more
  - Some argue it should now be 64kB or 256kB these days
- Hit time...
  - Call in 0
- Miss penalty...
  - Latency to get a block from disk: 1ms or so for an SSD...
    Or put in clock terms, a 1 GHz clock -> 1,000,000 clock cycles!

So what are the implications?

- As long as you don't really use it, Virtual Memory is great...
  - Basically as long as your ***working set*** fits in physical memory, virtual memory is great at handling a little extra...
- But as soon as your working set exceeds physical memory, your performance goes to crap...
  - The system starts ***thrashing***: Repeatedly needing to copy data to and from disk...
    - Similar to ***thrashing the cache*** when your working set exceeds cache capacity
  - If you have a spinning disk, it really starts sounding like the computer is suffering....

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

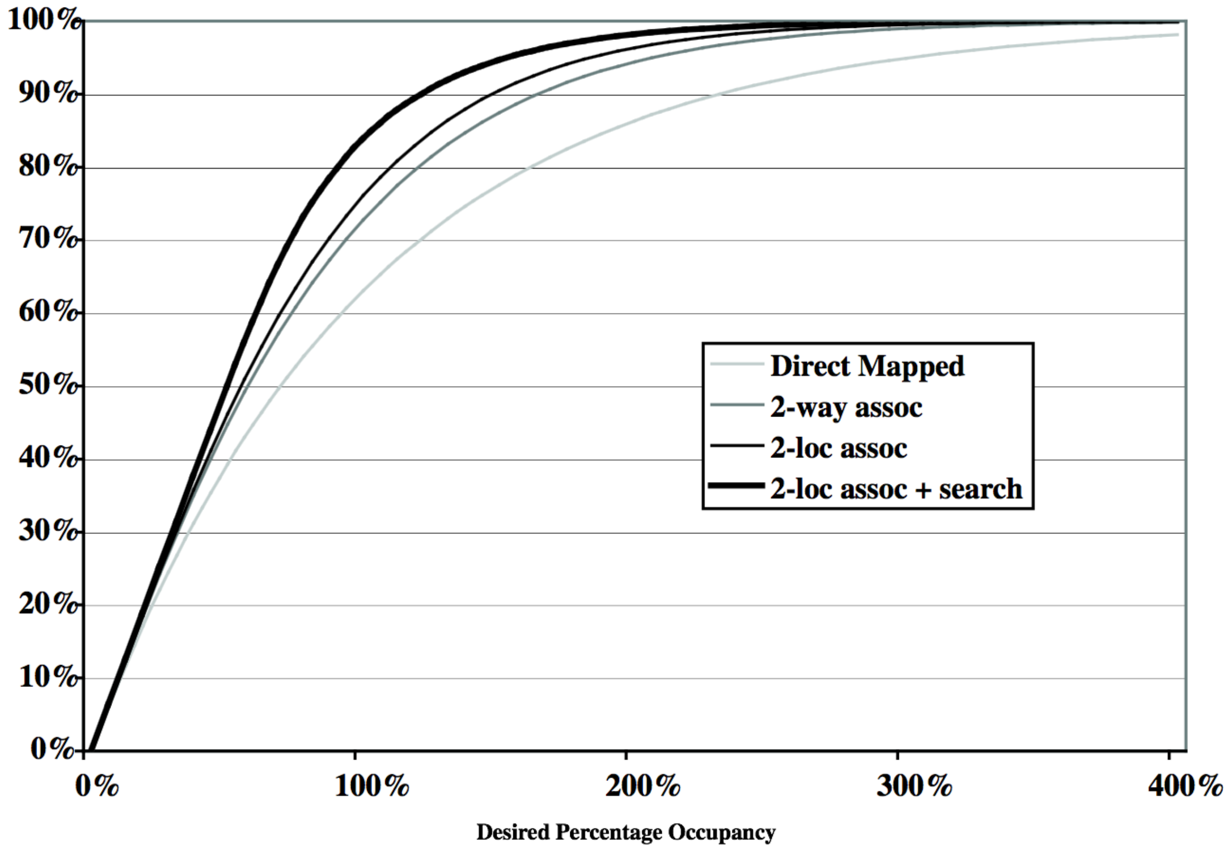
Nick's Cache #1: Permutation Cache

- 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