Virtual Memory → Advanced Caches → Memory

Topics:
- Improving AMAT
- PIPT cache
- VIPT cache
- Cache Aliasing
- Way Prediction (in SA Cache)
- NMRU + PLRU
- 3 C's
- Multi-Level Cache
- Local vs Global Hit Rate

Advanced Caches

Improving the performance of caches is done by improving the hit rate, improving the miss rate, or improving the miss penalty. The methods to implement these improvements in discussed in this lesson.

Improving Cache Performance

Three methods for improving cache performance can be derived from the AMAT equation:

AMAT = Hit time + Miss Rate * Miss Penalty (AMAT = Average Memory Access Time)

Improvement Methods:

- Reduce Hit Time
- Reduce Miss Rate
- Reduce Miss Penalty

THE FOLLOWING TECHNIQUES ARE USED FOR REDUCING THE HIT TIME.

Reduce Hit Time

We can also reduce Hit Time by making the Cache smaller, but this would make the Miss Rate higher.
We can reduce Hit Time by reducing the Cache Associativity (having less potential Cache Lines to search), but this also can cause a higher Miss Rate (because Blocks will have more conflicts & more likely to kick each other out, leading to more misses)

- Overlap cache hit with another hit (Via Pipeline Caches)
- Overlap cache hit with TLB hit
- Optimize Lookup for Common Case (without sacrificing the Uncommon Case too much)
- Maintain Replacement State Quicker (on hits, we need to update some state that we'll need later for replacements, and we can try to make this faster to reduce the hit time)

Overlap cache hit with another hit (to Reduce Hit Time)

Pipelined Caches

Pipelining caches = Overlapping a cache hit with another hit

Hit Time = Actual Hit + Wait Time

To pipeline a cache:

Partition the task into 3 stages:

- Stage 1 - Reading the index to find the set (from the Address)
- Stage 2 - Determining the hits and beginning the data read (determine a Hit by finding the Line in the Set with a Matching Tag and a Valid Bit = 1)
- Stage 3 - Finishing the data read (From the found Hit Cache Line, combine its data with the Block Offset to extract the data you want from the Cache Line.)

Overlap cache hit with TLB hit (to Reduce Hit Time)

TLB and Cache Hit

Remember that we first translate the Process's Virtual Address to a Physical Address, which we can then use with a "PIPT" cache. So if we want our Cache access to be fast (aka fast Hit Time), we'll want to have a TLB hit as well (since this needs to happen before we even access the Cache).

Cache Speed-up Method 2: Overlap the cache hit with a TLB hit.

A cache that uses the physical address is called a physically indexed-physically tagged cache (PIPT cache). With a PIPT cache, the TLB lookup/Virtual-to-Physical-Address Translation needs to occur FIRST before we can check the Cache (since it requires a Physical Address). But if we use a VIRTUALLY ACCESSED CACHE, then we don't need to wait for the Physical Address translation!

Virtually Accessed Cache

If the virtual address is used by the cache, then the TLB and the cache can both complete their tasks at the same time.

The downside of the virtually addressed cache is the virtual address is process specific. So the cache needs to be flushed with every context switch, leading to cache misses.

A second downside would be aliasing - leading to incorrect execution.

Initially, a Virtually Accessed Cache SEEMS to have the benefit of skipping the TLB access (which is needed to translate the Virtual Address to a Physical Address), however we still need to access the TLB (even on Cache Hits) because the TLB (actually the process's Page Table) contains additional data about Page PERMISSIONS to determine if we should even be permitted to read/write/execute that page of memory. So we can't claim that benefit of Virtually Accessed Caches, because TLB access is still required for Virtually Accessed Caches.

✳ Missing Notes that should go here are included at the end of the Lesson Notes.

aka VIPT Cache

Virtually Indexed - Physically Tagged Cache

Virtual Page #, the Index (which determines the Cache Set), and the Block Offset

Start with the virtual address. Partition into the ~~offset, index, and tag~~.

Index is used by the cache (Identifies the Set, within which we search the cache lines. These have the "Data", "Valid Bit", and "Tag"- which we'll use to check against the Physical Frame #)

Virtual Page # ~~Tag~~ is used get the frame number (from the TLB. If TLB miss, then go to Page Tables. Then combine the Frame Number with the Page Offset to get the Physical Address, which is then used to check with the "Tag" obtained from the Cache Line).

The tag check is used with the translated physical address. Note that we use the Physical Address instead of just the Frame Number bc a Page Frame is bigger than a Cache Line, so we need more bits of resolution.

Both the cache lookup and the translation are done at the same time. It can be done without having to flush the cache and without aliasing if the cache is small enough.

VIPT Cache Aliasing

Virtual Address → Physical Address

VA Page Number = PA Frame #

VA Page Offset = LSB of PA

There is no aliasing if all the index bits come from the page offset. The cache has to be small enough to do this. See the "Additional Notes" for a more detailed explanation for this.

Real VIPT Caches

Cache size must be less than or equal to the Associativity of the cache * Page Size

Ex: For a 4-Way SA Cache & a Page Size of 8 kB, our max VIPT cache size would be 32 kB (for no Aliasing).

Associativity and Hit Time

With higher associativity:

Improves the miss rate but makes the hit time longer (reduced miss rate caused by fewer conflicts, but having to search through a Set makes hits slower).

With Direct Mapped:

Improves the hit time but makes the miss rate worse Hit time is quick (less searching), but we are more likely to miss because of the increase in conflicts in the Cache.

Way Prediction

The processor knows the Set from the Index (from the Address). Way Prediction is trying to guess which LINE within a Set is most likely to be a hit.

With way prediction, the processor tries to guess which set is most likely to be a hit. If the prediction is correct the hit time goes down. If the prediction is wrong, then the normal set-associative check is used. If our prediction is correct, then our hit time comes equal to that of a Direct-Mapped Cache. If we're incorrect, then the hit time becomes the same as a normal Set-Associative Cache.

Way Prediction Performance

Using Way prediction the AMAT is better than with a direct mapped or a 8-way set associative cache.

Fully associative, 8-way set associative, and 2-way set associative can all benefit from way prediction.

Direct mapped caches do not benefit from way prediction. (while also suffering from the cost of a higher Miss Rate)

So Way Prediction (which is built on top of a Set-Associative Cache) reduces the Hit Time (with a correct prediction) while also gaining the benefits of a Set-Associative Cache (which reduces the Miss Rate).

Replacement Policy and Hit Time

Random selection has a good hit time, but poor miss rate. (good hit rate because nothing needs to be done on access, but poor miss rate bc it is likely to replace something that might been needed soon).

LRU has a good miss rate, but requires updates of all counters on a hit - slowing the hit time and increases power consumption. (LRU has good miss rate bc it is more likely to remove things that are less likely to be needed, but EACH access requires steps to maintain the LRU state, which reduces the Hit Time.

And remember, Hit Time is just the access time for ALL ACCESSES (HIT OR MISS)
Miss Time = Hit Time + Miss Penalty

Uses a single "pointer" PER SET to remember which line in that Set was the Most Recently Used. The space cost for that "pointer" depends on how many Lines are in a Set. For example, for a 2-Way SA Cache, you only need 1 bit per Set to track its NMRU (0 is first line, 1 is for second line). For a 4-Way SA Cache, you'll need 2 Bits Per Set (2^2 = 4).

NMRU Replacement Policy

NMRU = Not-Most Recently Used — Keeps the Miss Rate low while also doesn't add too much to the Hit Time.

To get a better replacement policy than LRU or Random use NMRU.

Just track the MRU and replace any other block.

PLRU Replacement Policy

PLRU - Pseudo LRU

PLRU Explanation: using 1 bit per line in a Set, start with all bits at 0. Whenever a replacement is needed to be made, randomly select one of the lines in the Set that has a 0 "PLRU bit". Whenever a Line in that Set is accessed, set that bit to 1. Continue with this until a final access causes the last Cache Line in that Set with a 0 "PLRU bit" to be set to 1. At that point, keep that Cache Line's "PLRU bit" set to 1, but reset all of the other Lines back to 0.

Every time a bit is accessed, set the LRU bit to 1. Kick out the LRU bit that equals 0. When all the LRU bits are 1, set the LRU block to 1 and reset all the LRU bits to 0.

PLRU is a compromise between LRU and NMLRU.

PLRU tries to more closely replicate LRU than the NMRU policy. It uses 1 BIT per cache line as opposed to having a single pointer for the NMRU, but also compared to having a COUNTER per line with LRU- so it's a middle ground compromise.
On a replacement, we need to find an entry with a 0 bit, kick it out, and update it from 0 to 1, whereas with the LRU we need to scan and update ALL the counters and with NMRU where we just change 1 pointer. So behavior is a sort of middle ground.

Reducing the AMAT

AMAT can be reduced by a lower hit time and a lower miss rate.

Causes of Misses:

3 Cs:

- Compulsory Miss - when a block is accessed for the first time — We'd still have this type of miss for an infinite cache.
- Capacity Miss - blocks evicted when the cache is full — even for Fully-Associative Caches (blocks can go to any Line).
- Conflict Miss - blocks are evicted due to associativity — (a miss that is not a Compulsory miss and also not a Capacity miss is then categorized as a Conflict Miss)

Larger Cache - reduces capacity misses

Larger Associativity - reduces conflict and some capacity misses

THE FOLLOWING TECHNIQUES ARE FOR REDUCING THE MISS RATE.

Larger Cache Blocks

Larger cache blocks bring more words in — into the Cache on a Cache Miss

- The miss rate improves with good spatial locality — More Cache Hits when we tend to use the extra words that we brought in.
- The miss rate degrades with poor spatial locality — More Cache Misses when we don't use the extra words that we brought in. Now they're just wasting cache space.

Increased block size can reduce compulsory, capacity, and conflict misses.

Larger Caches have more capacity, so they can tolerate bigger Cache Blocks, which decreases the miss rate for code with good spatial locality. When there's poor spatial locality, the "wasted" extra space brought in for a larger cache block won't be too bad relative to the large Cache size, so its okay. However, potential "waste" of using larger block sizes is more costly for Smaller Caches.

Prefetching

Prefetch blocks into the cache to improve the miss rate.

Good guesses eliminate misses, while bad guesses cause cache pollution.

Good guesses turn a potential miss into a Hit, while bad guesses pollutes the cache with something that isn't going to be used, so wastes cache space and causes a future miss (from kicking something out that could've been useful and needed for a Cache Hit).

Prefetching Instructions

The compiler can be used to determine which blocks to fetch.

The question becomes - how far in advance should the blocks be fetched?

This is a difficult question to answer with a compiler.

Hard pick a correct "prefetch distance" because it can be hardware dependent. "pdist" could be too big or too small. Fetch too late and you still have to suffer a miss. Fetch too early and its in the cache early, but it's at risk of getting replaced, or taking up space that causes another cache access to miss.

Hardware Prefetching

Hardware prefetching = using hardware to guess what will be accessed soon.

Types of Hardware prefetching:

- Stream buffer - prefetch the next block — Prefetches sequentially. Good for code with immediate spatial locality

Tries to observe spatial patterns of memory access and prefetch based on those distance patterns. For example, accessing blocks that are every "d" blocks away from it.

Stride prefetch - prefetch the block at distance 'd'

Correlating prefetcher - if A is fetched, then B should be prefetched, etc.

Good for temporal locality. For example, linked list nodes are not sequential in memory, but if you're iterating through the nodes, then a correlating prefetcher would be able to predict this memory access pattern. Sort of like a History Based Branch Predictor.

Loop Interchange

Loop interchange = compiler optimizations to change the code to have better locality by swapping the inner and outer loops to sequentially access the matrix in memory.

Basically, the compiler tries to make sure that if we are iterating through a matrix in memory, that the code iterates through the elements sequential (for good Spatial Locality)

THE FOLLOWING TECHNIQUES ARE USED FOR REDUCING THE MISS PENALTY.

Overlap Misses

Reducing the Miss Penalty can be accomplished by overlapping the misses.

While the OOO processor is waiting for a cache miss, it continues to execute instructions. Within these instructions, the processor may encounter another cache miss. This is an overlapping miss.

Blocking Cache: Only one load at a time can be performed. If a cache gets a load instruction, no other load instruction can be executed until the first load is completed.

Non-blocking Cache: a non-blocking cache will allow:

Hit-under-miss: if the processor is waiting for a miss, it can execute cache hits.
Miss-under-miss: if the processor is waiting for a miss, it can execute additional requests to memory (other misses).

The non-blocking cache can reduce the performance penalty for a miss by half, this ~~is~~ requries memory level parallelism.

Miss Under Miss Support in Caches

Miss Status Handling Registers (MSHR):

-keep information about misses that are in progress
-check MSHRs to see if the requested cache miss is one that has already been requested.

if the cache miss is a new miss (a true miss):

1. allocate an MSHR register
2. track which instruction is waiting for the miss

if the cache miss has already been requested (a Half-miss):

1. the instruction is added to the MSHR

When data arrives from memory:

-The MSHR is used to alert the correct instructions that their requested data is ready.
-Release the MSHR register

How many MSHR registers are necessary?

16 - 32 MSHR is a typical number

Applications that can benefit from the MSHR are those that have misses often enough so the next miss can be issued and executed before the processor runs out of resources while "stalling" on the previous miss. So if your application only has a miss every 1000 instructions, then it wouldn't benefit form a MSHR.

Cache Hierarchies aka Multi-Level Caches

Different level caches are used to reduce the AMAT specifically, to reduce the Miss Penalty

If there is a miss in the first level cache, a second level cache is checked. If this level is a hit, then the time spent going to main memory is saved.

L1 miss penalty = L2 hit time + L2 miss rate * L2 miss penalty

AMAT with Cache Hierarchies

AMAT = L1 hit time + L1 miss rate*L1 miss penalty
L1 miss penalty = L2 hit time + L2 miss rate * L2 miss penalty
L2 miss penalty = L3 hit time + L3 miss rate * L3 miss penalty

The equations will continue until the L# miss penalty = the main memory latency. This is called the Last Level Cache (LLC)
L1 capacity < L2 capacity
L1 latency < L2 latency

Multi-Level Cache Performance

Having a cache is better than no cache, and having a 2 level cache is better than having a one level cache. Each level reduces the Miss Penalty, which means it reduces the AMAT.

Hit Rate A Local Hit Rate is a hit rate that the cache actually observes. For L2, L3, etc., their local hit rate is lower because they usually DON'T see the easy accesses (which is taken care of by the L1 cache)

~~Local hit = a hit to an individual cache. For example a local hit for the L2 cache is a hit in the L2 cache.~~

~~Global hit = the hits to all the caches.~~

So when we talk about the cache size and how bigger caches behave better and so on as far as hit rate is concerned, usually, we think about the Global Hit Rate, because the local hit rate heavily depends on what do you have as the level 1 cache. Its global hit rate is going to be similar to our hit rate of that type of a cache if it was used alone.

Global Vs Local Hit Rate

Global Hit Rate = 1 - Global Miss Rate

Global Miss Rate = # of Misses in this cache / # of all memory accesses ALL memory accesses, not just the ones that reach this Cache.

Local Hit Rate = # of Hits / # of Access to this cache

MPKI = Misses per 1000 instructions This is similar to the Global Miss Rate, except that it doesn't normalize the misses with the number of just memory accesses, it normalizes with the number of instructions.

Inclusion Property

There are 3 different possibilities for the same block being in L1 and L2:

1. A block in L1 may or may not be in L2
2. A block in L1 must also be in L2 (Inclusion)
3. A block in L1 cannot also be in L2 (Exclusion)

If inclusion or exclusion is not enforced, then the block may or may not be in L2

To enforce inclusion an inclusion bit is required. The bit will track if a block is in the other level caches. For example, to enforce Inclusion for an L2 cache (meaning that everything in the L1 cache MUST also be included in the L2 cache - they cannot be replaced), we add an extra bit to mark those Memory Blocks in the L2 Cache to prevent them from being replaced until they're removed from the L1 Cache.

Aliasing in Virtually Accessed Caches

Aliasing is a big problem in virtually addressed caches.

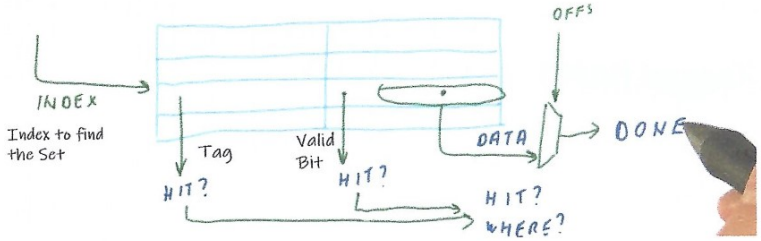
Aliasing will result in incorrect execution. So every write to the cache must check for aliasing, this will lead to a degradation in performance.

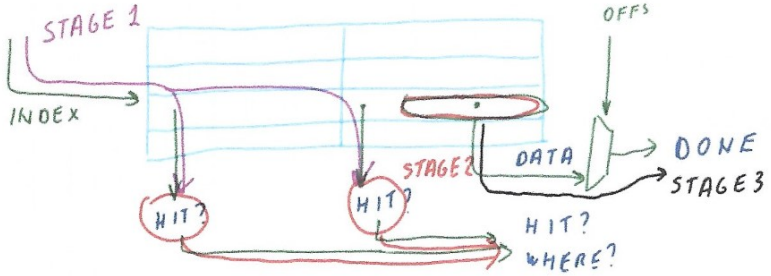
Additional Notes

Pipelined Caches (Reduce Hit Time) Example

PIPELINED CACHES

- MULTIPLE CYCLES TO ACCESS
  - ACCESS COMES IN CYCLE N (HIT)
  - ACCESS COMES IN CYCLE N+1 (HIT) HAS TO WAIT
- HIT TIME = ACTUAL HIT + WAIT TIME

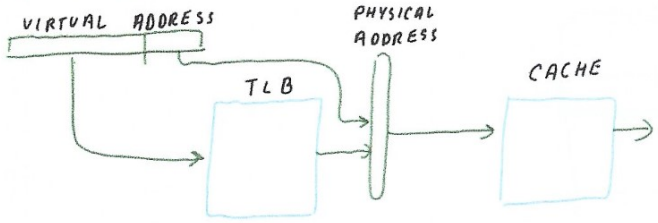




Reduce Hit Time by Overlapping Cache Hit w/ TLB Hit

PIPT Cache

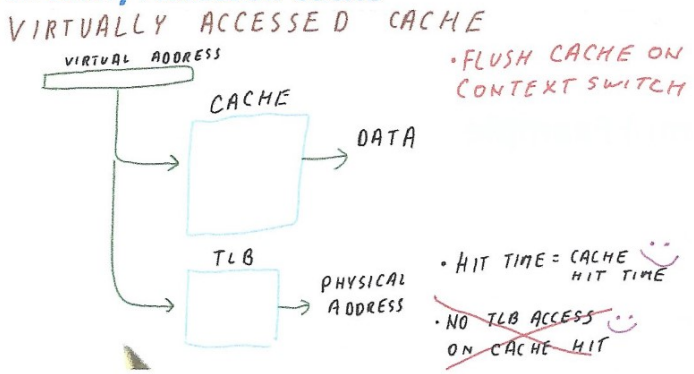
TLB AND CACHE HIT



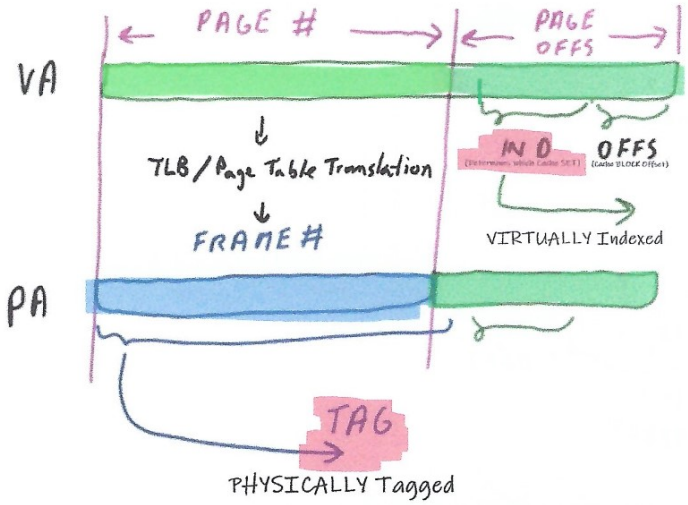
- PHYSICALLY ACCESSED CACHE
- PHYSICAL CACHE
- PHYSICALLY INDEXED - PHYSICALLY TAGGED CACHE

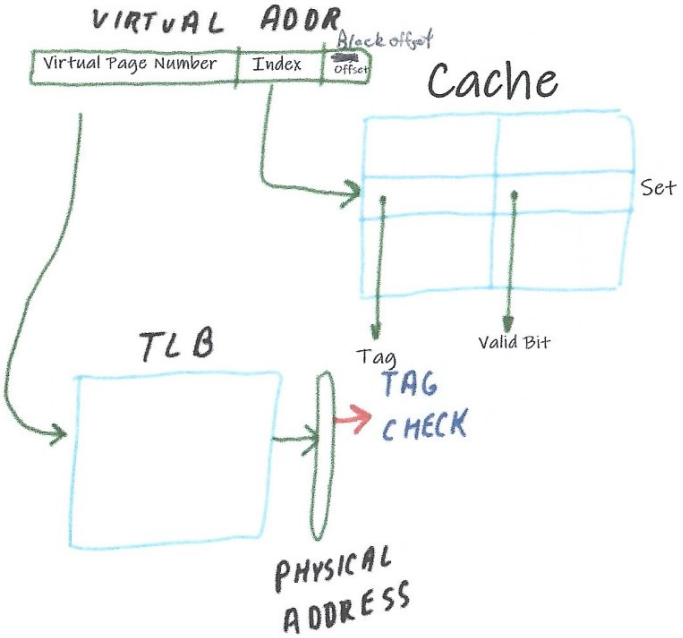
PIPT

Virtually Accessed Cache



Virtually Indexed, Physically Tagged (VIPT)



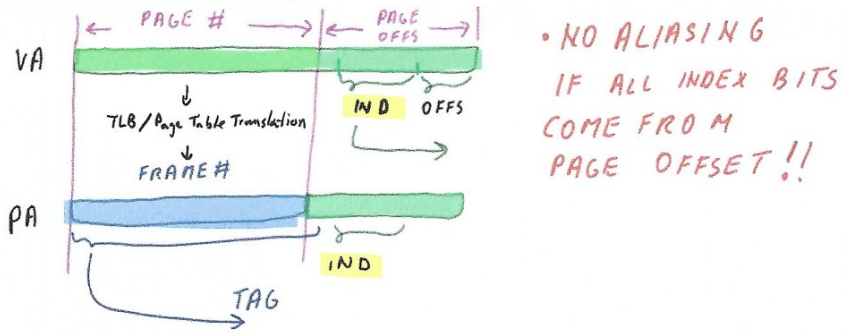


You use the "Index" to find the Set in the Cache, and within that Set, you get the Tag & Valid Bit for each Cache Line in the Set. AT THE SAME TIME (aka in parallel), you take the Virtual Page Number from the Virtual Address and you search for a matching entry in the TLB (if TLB Miss, then go to Page Table(s) for V->P Translation). You then extract the Tag from the Physical Address and then compare that with the Tag that you got from the Cache Set.

Aliasing in Virtually Accessed Caches

It basically happens when two Virtual Addresses in the same Virtual Address Space (aka in the same process) both map to the same Physical Address (which is possible, for example, when the same process opens the same file twice or something). However, these two Virtual Addresses will then map to two different Cache Locations, which can cause issues because then there's no longer a singe point of truth between the two virtual addresses.

VIPT Cache Aliasing



Notice here how (if the number of Index bits are small enough) all of the Index bits can fit within the "Page Offset" section, which doesn't change between the Virtual Address and the Physical Address.

So, although we are indexing using the virtual address, in reality, we are actually using the same index that we would be using if we were using the physical address. Which means, there will be no aliasing. How is there no aliasing? Well, because the virtual pages that have **different Virtual Page Numbers** that map to the ***same* Physical Frame Number** can only differ in the page number, but they have to have the ***same* Offsets for the *same* data**. And because only the Page Offset matters for the index, if the index is low enough here, what happens here is that all of the data that can possibly be in the same place in the Physical memory **has to be in the same Set**, because the index is determined only from the part that has to be the same. So we have that there is no aliasing if all of the index bits that we use came from the Page Offset, because really those are the same index bits that would come from the physical address if we had the Physically Indexed Cache. We like this very much, but the cache has to be small enough to do this.

> Remember that we determine the "Cache Set" from the Index.
> **Explanation**: So if we have two separate Virtual Addresses that map to the same Physical Address space, they do this by mapping to the same PHYSICAL FRAME. So this means that these two separate Virtual Addresses would differ by having different Virtual Page Numbers, but their respective Page Offsets are both Offsets of the SAME PHYSICAL FRAME. So if two different virtual addresses were *really* referencing the same physical data, then the Virtual Page Number portion of their address could differ (they'd end up be translated to the same Physical Frame Number), but their page offsets would be the same, which means that (if the

Index bits all fit within the Page Offset portion of the address) their Index bits would be the same, which means that they'd use the same Cache set!

VIPT ALIASING AVOIDANCE QUIZ

- 4-WAY SET-ASSOCIATIVE
- 16-BYTE BLOCK SIZE
- 8kB PAGE SIZE

NO ALIASING ⇒ MAX SIZE IS 32 kB

16 B block size → 2^4 → 4 bits for Block Offset (Cache)

8kB Page Size → $2^3 \cdot 2^{10}$ → 13 bits for Page Offset (Page)

So we'd want (13-4) = 9 bits max to use for our index.

Since our cache is 4-way Set-Associative, each one of our Index combinations would identify a Set with 4 cache Lines.

So how many cache lines would we have?

(# of Sets)(# cache lines per set) = $(2^9)(4) = (2^9)(2^2) = 2^{11}$ cache lines

So how big is this cache?

(# of cache lines)(Block size per Cache Line) = $(2^{11})(16B)$

$= (2^{11})(2^4 B)$

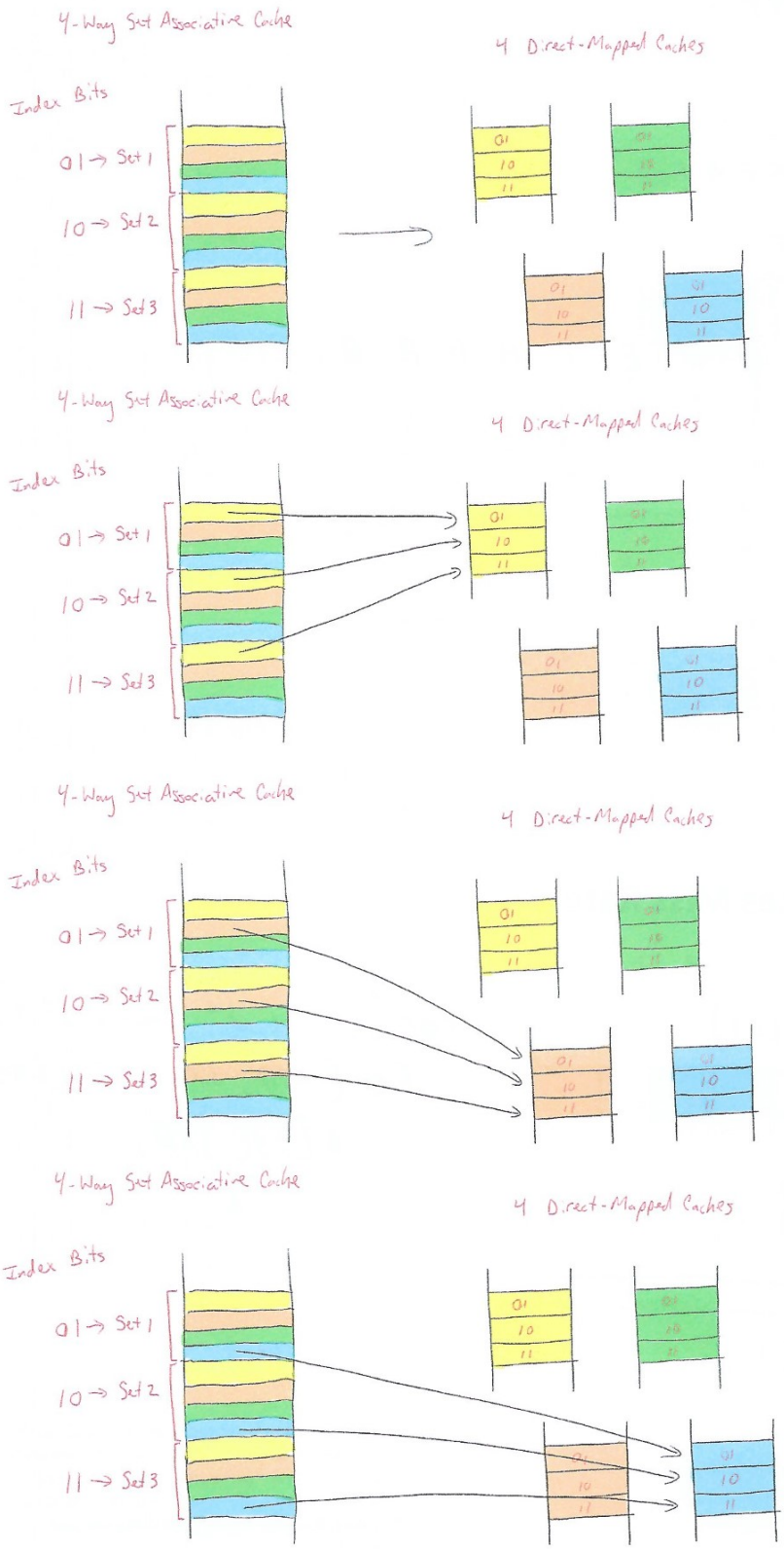
$= (2^{15}\ kB)$

$= 32kB$

Way Prediction

So with Way Prediction, we're sorta picking one of the Lines within a Set.

So like let's say if we have a 4-Way Set-Associative Cache, that means that out of every Set, there are 4 cache lines that we need to check. So we can say that we have Sets of 4. We can think of splitting up each Set and combine their respective components together (like gather all the 1st element of each Set together, gather all the 2nd elements of each Set together, all the 3rd elements together, and all the 4th elements together), each of these combinations could be seen as a Direct-Mapped cache. So a 4-Way Set Associative Cache could be seen as 4 Direct-Mapped Caches. So with Way Prediction, we're going to "Predict" which one of these 4 **"logical" Direct-Mapped Caches** we're going to use. In other words, **out of 4 "ways" in a Set, we're going to predict which "way" to use**. So previously, where the index bits would give us a set of 4 Cache lines to check, with a Direct-Mapped Cache, the index bits tell us exactly 1 Cache Line to check. If it's a Hit, then we get the good Hit Time of a Direct Mapped Cache. If it's a miss, then we check the 4-Way Set Associative Cache as normal (by checking the other 3 Cache lines within that Set), so on a wrong prediction the Hit Time is like a normal 4-Way Set Associative Cache. With **Way Prediction**, we get the benefits of potentially predicting the correct "way" within a Set and getting a **faster Hit Time (comparable to a Direct-Mapped Cache)** while still keeping the benefit of a **Low Miss Rate** that Associative Caches get (which Direct-Mapped Caches don't have).


4-Way Set Associative Cache
4 Direct-Mapped Caches
Index Bits
01 → Set 1
10 → Set 2
11 → Set 3
01
10
11
4-Way Set Associative Cache
4 Direct-Mapped Caches
Index Bits
01 → Set 1
10 → Set 2
11 → Set 3
01
10
11
4-Way Set Associative Cache
4 Direct-Mapped Caches
Index Bits
01 → Set 1
10 → Set 2
11 → Set 3
01
10
11
4-Way Set Associative Cache
4 Direct-Mapped Caches
Index Bits
01 → Set 1
10 → Set 2
11 → Set 3
01
10
11

NMRU Replacement Policy

NMRU QUIZ

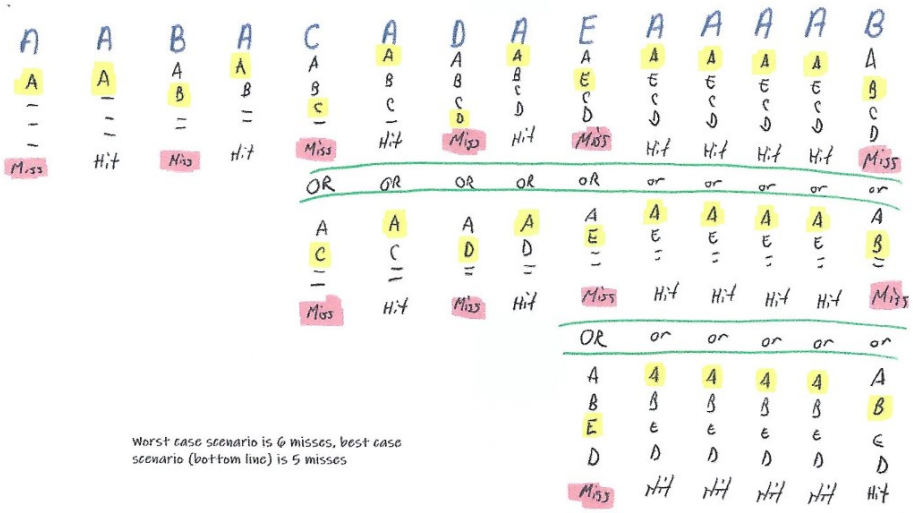
- FULLY ASSOC CACHE WITH 4 LINES
- NMRU REPLACEMENT
- STARTS OUT EMPTY

PROCESSOR ACCESSES BLOCKS

A A B A C A D A E A A A A B

WE HAVE AT LEAST 5 MISSES

AT MOST 6 MISSES



Worst case scenario is 6 misses, best case scenario (bottom line) is 5 misses

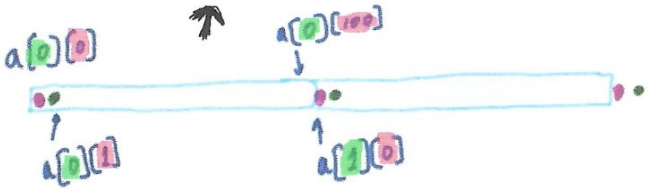
Loop Interchange (Reduces Miss Rate)

LOOP INTERCHANGE

```
for (i=0; i< ~ ; i++)
    for(j=0, j< ~ ; j++)
        a[j][i] = 0;
```

=>

```
for (j=0; j< ~ ; j++)
    for(i=0; i< ~ , i++)
        a[j][i] = 0;
```



- NOT ALWAYS POSSIBLE!

The access pattern of this array is not sequential. For every inner loop iteration, we will increment J, which will cause us to jump across memory, potentially into another Memory Block (which could be a Cache Miss/Conflict)

When the Compiler transforms this loop with Loop Interchange, the array access pattern is changed so that we're accessing the array memory sequentially, which improves Spatial Locality, leading to less Cache Misses and Conflicts.

Multi-Level Cache Performance

MULTI-LEVEL CACHE PERFORMANCE

	16 kB	128 kB	NO CACHE	L1 = 16 kB L2 = 128 kB
HIT TIME	2	10	100	2 FOR L1 12 FOR L2
HIT RATE	90%	97.5%	100%	90% FOR L1 75% FOR L2
AMAT	2 + 0.1 × 100 12	10 + 0.025 × 100 12.5	100	5.5

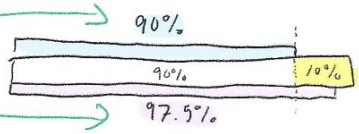
$2 + 0.1 \times (10 + 0.25 \times 100)$

35

35 is a much better "L1 Miss Penalty" than 100

"Global Hit Rates"

How'd we get 75%?



This is defined as the "Local Hit Rate"

The 97.5% L2 coverage overlaps w/ the 90% L1 coverage, so for the remaining 10% that is not covered by the L1 cache is covered by the L2 cache, which means that $\frac{7.5}{10}$ out of the accesses that get to the L2 cache (which is 10% of all accesses), the L2 cache has a hit rate of 75%.

Global & Local Hit Rates

GLOBAL AND LOCAL MISS RATE QUIZ

- L1 CACHE HAS 90% HIT RATE
  - LOCAL MISS RATE IS 10%
  - GLOBAL MISS RATE IS 10%

100% - 90% = 10%

- L2 CACHE HITS FOR 50% OF L1 MISSES
  - LOCAL MISS RATE IS 50%
  - GLOBAL MISS RATE IS 5%

Considering only the memory accesses that reach the L2 Cache, what percentage of them are misses? Since we get all L1 misses, and from that amount, we hit 50% of the time, that means we have a Local Miss Rate of 50%.

Out of ALL memory accesses, what is the percentage that our L2 cache misses? So if 10% of all L1 Cache accesses miss and go to the L2 cache, and then the L2 Cache misses 50% of those, then we have a Global Miss Rate of 10% * 50% = 5%.