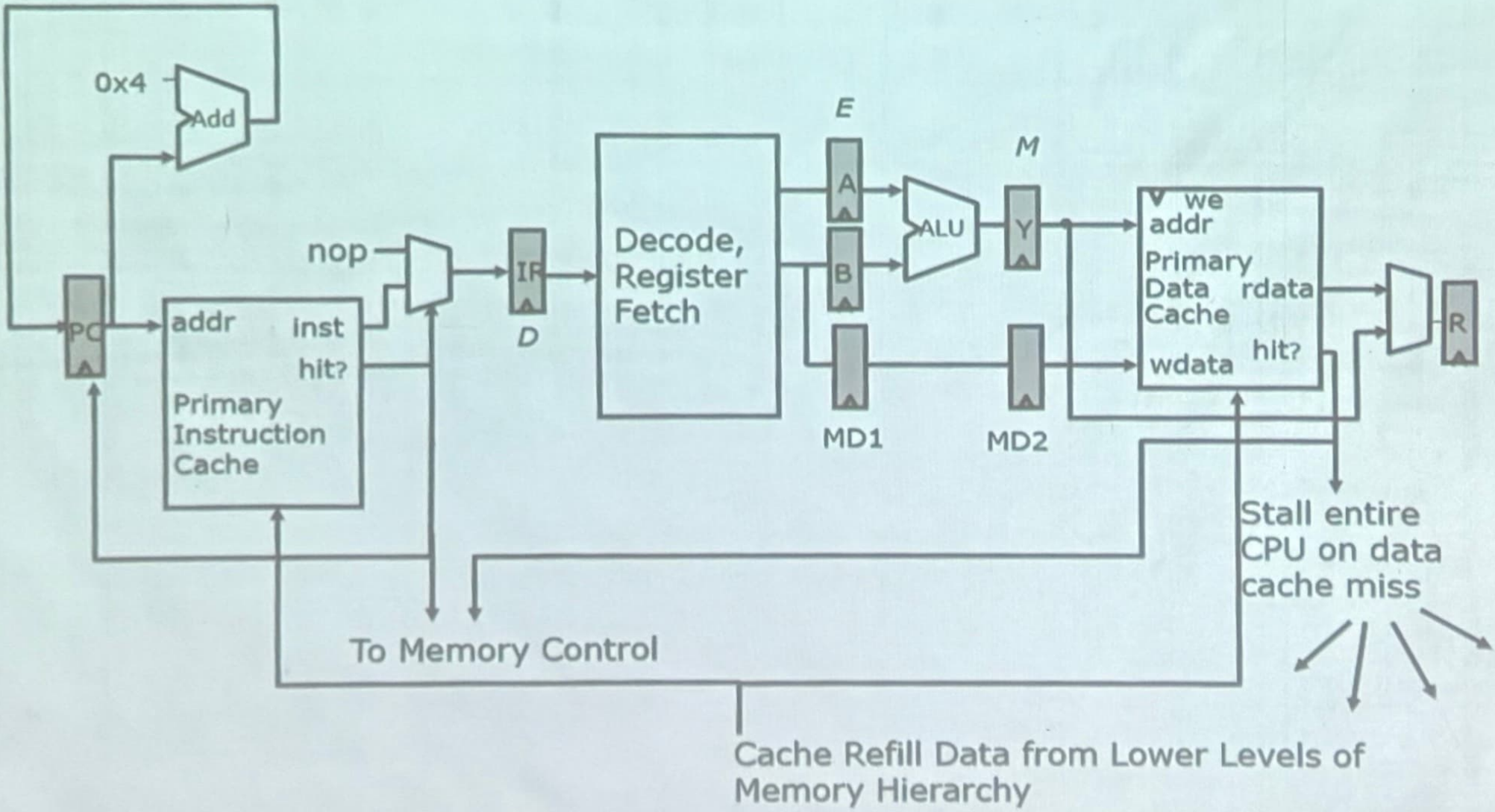
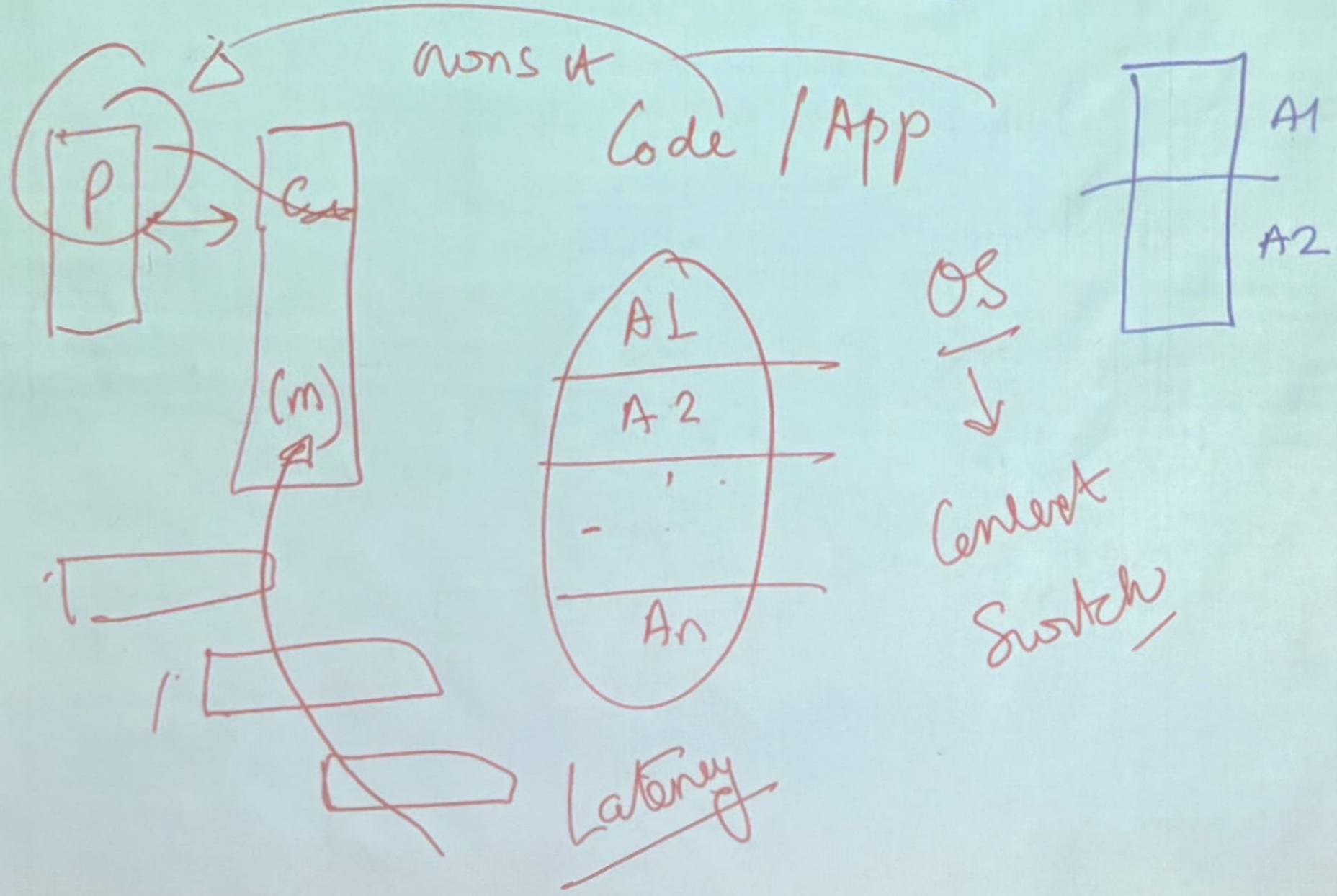
Datapath – Cache Interaction



Types of Cache Misses

- Compulsory misses: happens the first time a memory word is accessed – the misses for an infinite cache
- Capacity misses: happens because the program touched many other words before re-touching the same word – the misses for a fully-associative cache
- Conflict misses: happens because two words map to the same location in the cache – the misses generated while moving from a fully-associative to a direct-mapped cache
- Sidenote: can a fully-associative cache have more misses than a direct-mapped cache of the same size?

A1 ⟶ A2



More Cache Basics

- L1 caches are split as instruction and data; L2 and L3 are unified
- The L1/L2 hierarchy can be inclusive, exclusive, or non-inclusive
- On a write, you can do write-allocate or write-no-allocate
- On a write, you can do writeback or write-through; write-back reduces traffic, write-through simplifies coherence
- Reads get higher priority; writes are usually buffered
- L1 does parallel tag/data access; L2/L3 does serial tag/data

Average Memory Access Time

AMAT = [Hit Time] + [Miss Prob.] x [Miss Penalty]

- Miss Penalty equals AMAT of next cache/memory/storage level.
- AMAT is recursively defined

To improve performance

- Reduce the hit time (e.g., smaller cache)
- Reduce the miss rate (e.g., larger cache)
- Reduce the miss penalty (e.g., optimize the next level)

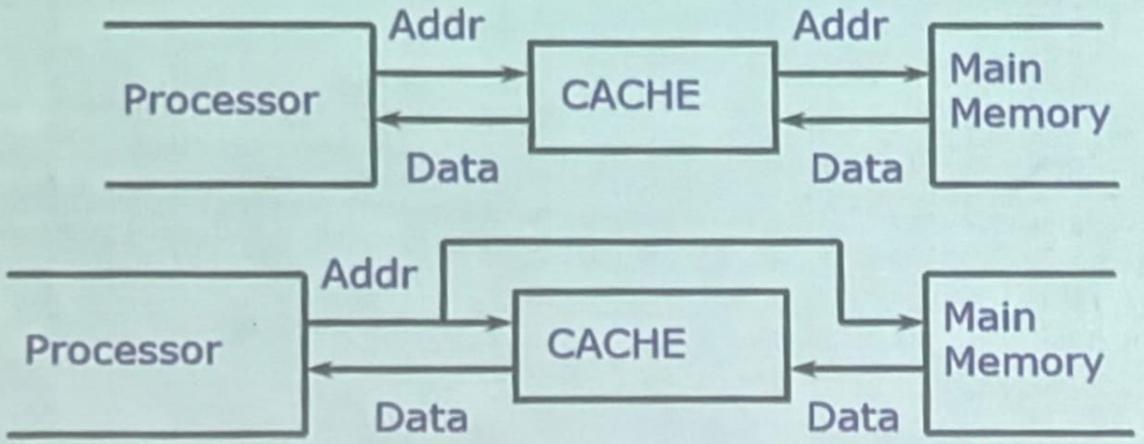
Simple design strategy

- Observe that hit time increases with cache size
- Design the largest possible cache with a hit time of 1-2 cycles.
- For example, design 8-32KB of cache in modern technology
- Design trade-offs are more complex with superscalar architectures and multi-ported memories

Example: Serial vs Parallel Access

Serial Access

- Check cache for addressed data. If miss (probability 1-p), go to memory
- AMAT = Tcache + (1-p) x Tmem



Parallel Access

- Check cache and memory for addressed data.
- AMAT = p x Tcache + (1-p) x Tmem
- AMAT reductions from parallel access often small. Tmem >> Tcache and p is high. Parallel access consumes memory bandwidth Parallel access increases cache complexity and Tcache.

Cache Misses and Causes (3C's)

Compulsory

- First reference to a block. May be caused by "cold" caches as application begins execution.
- Compulsory misses would occur even with infinitely sized cache

Capacity

- Cache is too small and cannot hold all data needed by the program.
- Capacity misses would occur even under perfect replacement policy.

Conflict

- Cache line replacement policy causes collisions
- Conflict misses would not occur with full associativity

Tolerating Miss Penalty

- Out of order execution: can do other useful work while waiting for the miss – can have multiple cache misses -- cache controller has to keep track of multiple outstanding misses (non-blocking cache)

- Hardware and software prefetching into prefetch buffers – aggressive prefetching can increase contention for buses

Cache Performance

Larger Cache Size

- Benefit: reduces capacity and conflict misses
- Cost: increases hit time

Higher Associativity

- Benefit: reduces conflict misses
- Cost: increases hit time

Larger Line Size

- Benefit: reduces compulsory and capacity misses
- Cost: increases conflict misses and increases miss penalty

Reducing Miss Rate

- Large block size – reduces compulsory misses, reduces miss penalty in case of spatial locality – increases traffic between different levels, space waste, and conflict misses
- Large cache – reduces capacity/conflict misses – access time penalty
- High associativity – reduces conflict misses – rule of thumb: 2-way cache of capacity N/2 has the same miss rate as 1-way cache of capacity N – more energy

Techniques to Reduce Cache Misses

- Victim caches
- Better replacement policies – pseudo-LRU, NRU, DRRIP
  -- insertion, promotion, victim selection
- Prefetching, cache compression

Victim Caches

- A direct-mapped cache suffers from misses because multiple pieces of data map to the same location
- The processor often tries to access data that it recently discarded – all discards are placed in a small victim cache (4 or 8 entries) – the victim cache is checked before going to L2
- Can be viewed as additional associativity for a few sets that tend to have the most conflicts

Replacement Policies

- Pseudo-LRU: maintain a tree and keep track of which side of the tree was touched more recently; simple bit ops
- NRU: every block in a set has a bit; the bit is made zero when the block is touched; if all are zero, make all one; a block with bit set to 1 is evicted

Cache Write Policy Alternatives

What happens when a cache line is written?

If write hits in cache (i.e., line already cached)

- Write-Through: Write data to both cache and memory. Increases memory traffic but allows simpler datapath and cache controllers.
- Write-Back: Write data to cache only. Write data to memory only when cache line is replaced (e.g., conflict).
- Write-Back Optimization: Insert "dirty bit" per cache line, which indicates whether line is modified. Write-back only if replaced cache line is dirty.

If write misses in cache

- No Write Allocate: Write data to memory only.
- Write Allocate: Fetch data into cache. Write data into cache. Also known as fetch-on-write.

Common combinations

- Write-through and no-write allocate
- Write-back and write allocate.

Reducing Write Time

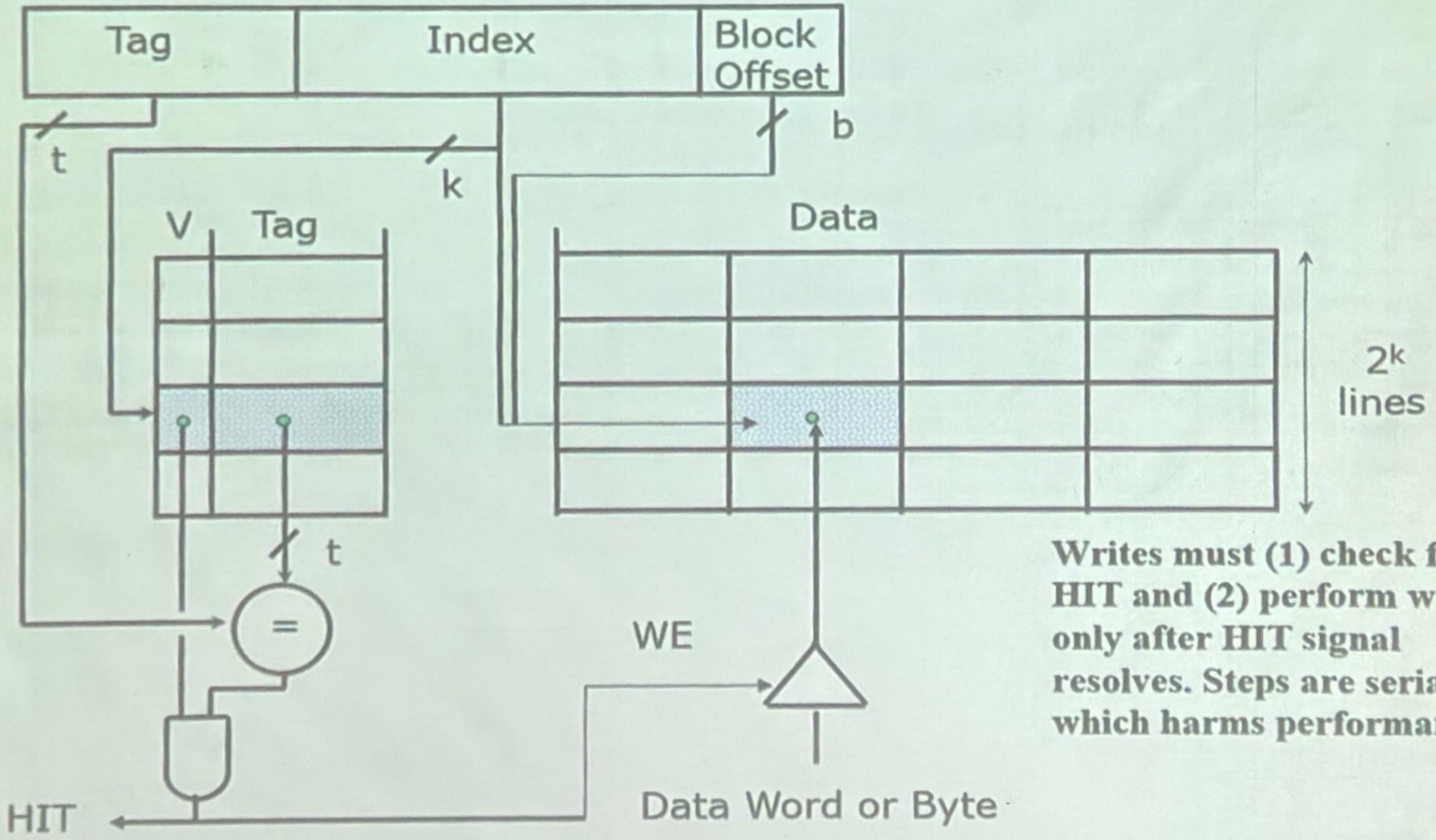
Problem: Writes take two cycles in memory

- Access cache and compare tags to generate HIT signal (1 cycle)
- Perform cache write if HIT enables a write (1 cycle)

Solutions

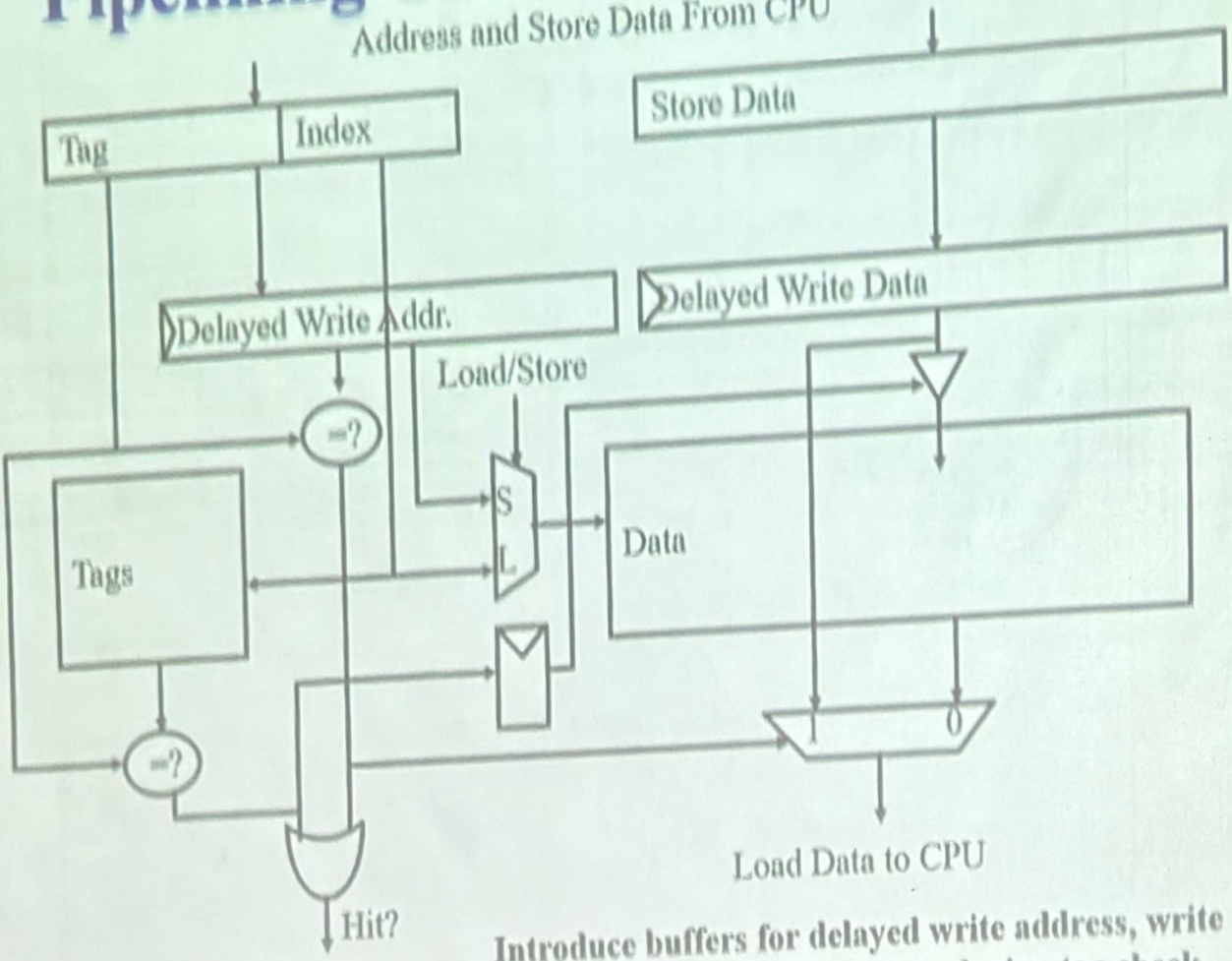
- Design data SRAM that can perform read/write in one cycle, restoring old value if HIT is false.
- Design content-addressable data SRAM (CAM), which enables word line only if HIT is true.
- Pipeline writes with a write buffer. Write the cache for j-th store instruction during the tag check for (j+1)-th store instruction

Write Performance



Writes must (1) check for HIT and (2) perform write only after HIT signal resolves. Steps are serial, which harms performance.

Pipelining Cache Writes



Introduce buffers for delayed write address, write data. Write cache for j-th store during tag check for (j+1)-th store. Note that loads must check buffers for latest data.

Buffering Writes

With buffers, writes do not stall datapath

- Introduce a write buffer between adjacent levels in cache hierarchy.
- After writes enter buffer, computation proceeds.
- For example: Reads bypass writes.

Problem

- Write buffer may hold latest value for a read instruction's address

Solutions

- Option 1: If a read misses in cache, wait for the write buffer to empty
- Option 2: Compare read address with write buffer addresses. If no match, allow read to bypass writes. Else, return value in write buffer.

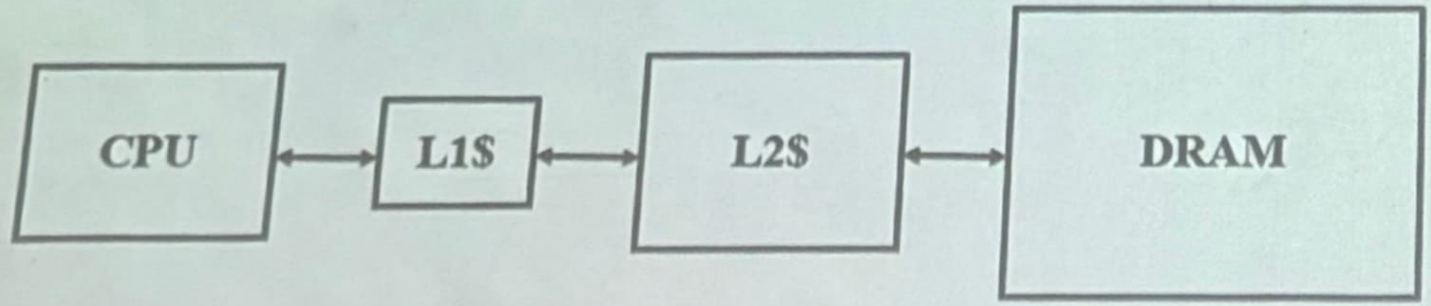
Cache Hierarchy

Problem

- Memory technology imposes trade-off between speed and size.
- A memory cannot be both large and fast.

Solution

- Introduce a multi-level cache hierarchy.
- As distance from datapath increases, increase cache size.



L1-L2 Cache Interactions

Use smaller L1 cache if L2 cache is present

- Reduce L1 hit time, but increase L1 miss rate.
- L2 mitigates higher L1 miss rate by reducing L1 miss penalty
- May reduce average time (AMAT) and energy (AMAE)

Use write-through L1 if write-back L2 cache is present

- Write-through L1 simplifies pipeline, cache controller.
- No write-backs for dirty lines reduces complexity.
- Write-back L2 absorbs write traffic. Writes do not go off-chip to DRAM.

Inclusion Policies

- Inclusive Multi-level Cache: Smaller cache (e.g., L1) holds copies of data in larger cache (e.g., L2). Simpler policies.
- Exclusive Multi-level Cache: Smaller cache (e.g., L1) holds data not in larger cache (e.g., L2). Example: AMD Athlon with 64KB primary and 256KB secondary. Reduces duplication.

Prefetching

- Hardware prefetching can be employed for any of the cache levels
- It can introduce cache pollution – prefetched data is often placed in a separate prefetch buffer to avoid pollution – this buffer must be looked up in parallel with the cache access
- Aggressive prefetching increases "coverage", but leads to a reduction in "accuracy" → wasted memory bandwidth
- Prefetches must be timely: they must be issued sufficiently in advance to hide the latency, but not too early (to avoid pollution and eviction before use)

Prefetching

Speculate about future memory accesses

- Predict likely instruction and data accesses.
- Pre-emptively fetch instructions and data into caches.
- Instructions accesses likely easier to predict than data accesses.
- Mechanisms might be implemented in HW, SW or both.
- What type of misses does prefetching affect?

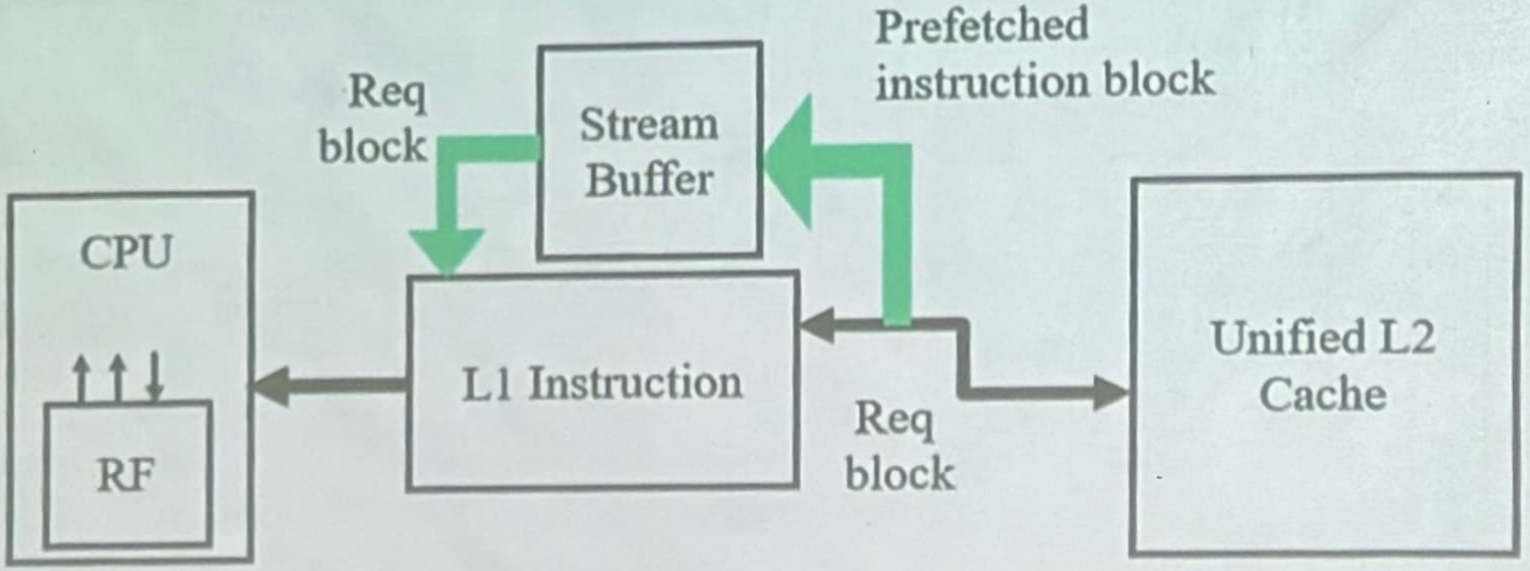
Challenges in Prefetching

- Prefetching should be useful. Prefetches should reduce misses.
- Prefetching should be timely. Prefetches will pollute the cache if too early and will be useless if too late.
- Prefetching may pollute caches and consume memory bandwidth.

Hardware Instruction Prefetching

Example: Alpha AXP 21064

- Prefetch instructions
- Fetch two lines on a cache miss. Fetch the requested line (i) and the next consecutive line (i+1).
- Place requested line in instruction L1 cache. Place next line in an instruction stream buffer.
- If an instruction fetch misses in L1 cache but hits in stream buffer, move stream buffer line into L1 cache. And prefetch next line (i+2)



Hardware Data Prefetching

Prefetch after a cache miss

- Prefetch line (i+1) if an access for line (i) misses in the cache

One Block Look-ahead (OBL)

- Blocks also known as lines.
- Initiate prefetch for block (i+1) when block (i) is accessed.
- Generalizes to N-block look-ahead
- How is this different from increasing the block or line size by N times?

Strided Prefetch

- Observe sequence of accesses to cache lines.
- Suppose a sequence (i), (i+N), (i+2N) is observed. Prefetch (i+3N).
- N is the stride.
- Example: IBM Power 5 (2003) supports eight independent streams of strided prefetches per processor, prefetching 12 lines ahead of the current access.

Software Prefetching

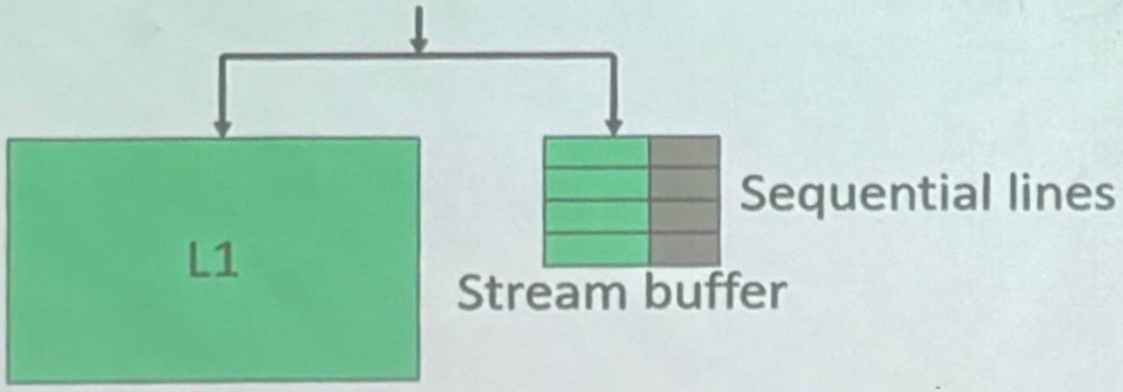
```
for(i=0; i < N; i++) {
  prefetch( &a[i + P] );
  prefetch( &b[i + P] );
  SUM = SUM + a[i] * b[i];
}
```

Challenges in Software Prefetching

- Timing is the biggest difficulty, not predictability.
- Prefetch too late if prefetch instruction too close to data use.
- Prefetch too early and cause cache/bandwidth pollution.
- Requires estimating prefetch latency: time from issuing prefetch to filling L1 cache line.
- Why is this hard to do? What is the correct value of P above?

Stream Buffers

- Simplest form of prefetch: on every miss, bring in multiple cache lines
- When you read the top of the queue, bring in the next line



Caches and Code

Restructuring code affects data access sequences

- Group data accesses together to improve spatial locality
- Re-order data accesses to improve temporal locality

Prevent data from entering the cache

- Useful for variables that are only accessed once
- Requires SW to communicate hints to HW.
- Example: "no-allocate" instruction hints

Kill data that will never be used again

- Streaming data provides spatial locality but not temporal locality
- If particular lines contain dead data, use them in replacement policy.
- Toward software-managed caches

Loop Interchange

```
for(j=0; j < N; j++) {
   for(i=0; i < M; i++) {
      x[i][j] = 2 * x[i][j];
   }
}
```

```
for(i=0; i < M; i++) {
   for(j=0; j < N; j++) {
      x[i][j] = 2 * x[i][j];
   }
}
```

What type of locality does this improve?

What does it assume about x?

Loop Fusion

```
for(i=0; i < N; i++)
    a[i] = b[i] * c[i];

for(i=0; i < N; i++)
    d[i] = a[i] * c[i];
```

What type of locality does this improve?

Loop Fusion

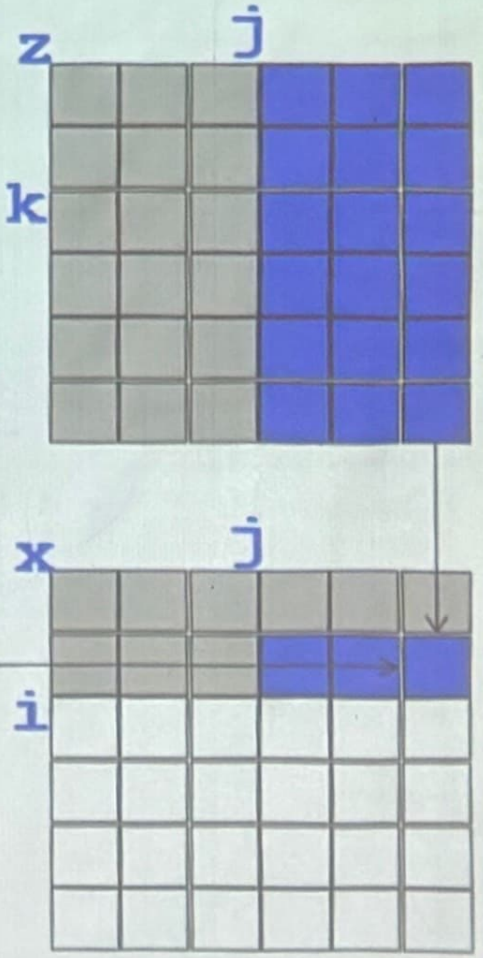
```
for(i=0; i < N; i++)
    a[i] = b[i] * c[i];

for(i=0; i < N; i++)
    d[i] = a[i] * c[i];
```

```
for(i=0; i < N; i++)
{
        a[i] = b[i] * c[i];
        d[i] = a[i] * c[i];
}
```

What type of locality does this improve?

Matrix Multiply (X=YZ) – Naïve

```
for(i=0; i < N; i++)
      for(j=0; j < N; j++) {
          r = 0;
          for(k=0; k < N; k++)
            r = r + y[i][k] * z[k][j];
          x[i][j] = r;
      }
```



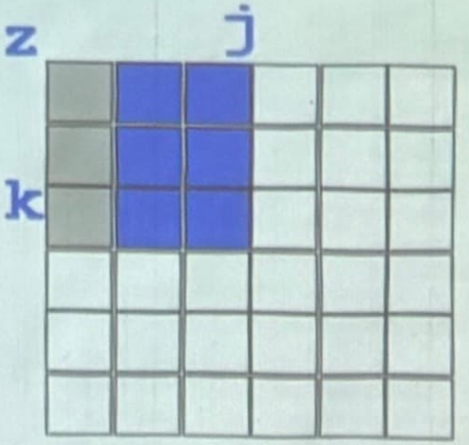
Notes

1. Iterate through matrix (y) by row.
2. Iterate through matrix (z) by col.
3. Update matrix (x) by col.

Not touched Old access New access

Matrix Multiply (X=YZ) – Blocked

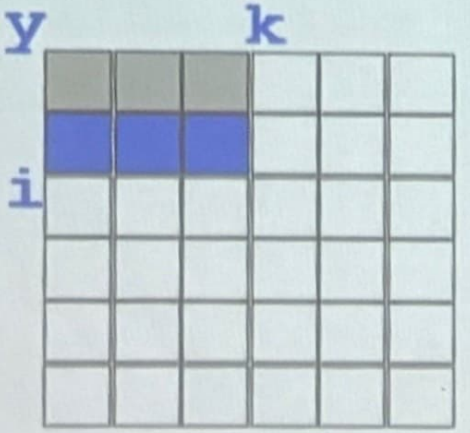
```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
            }
```

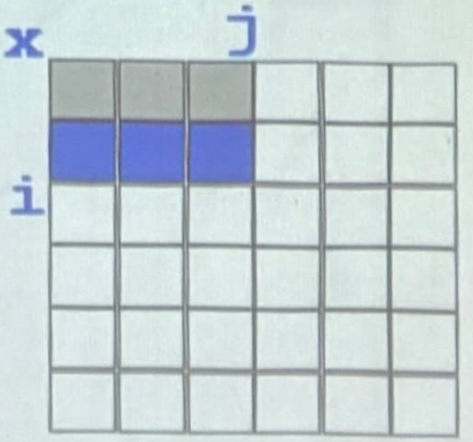


Notes

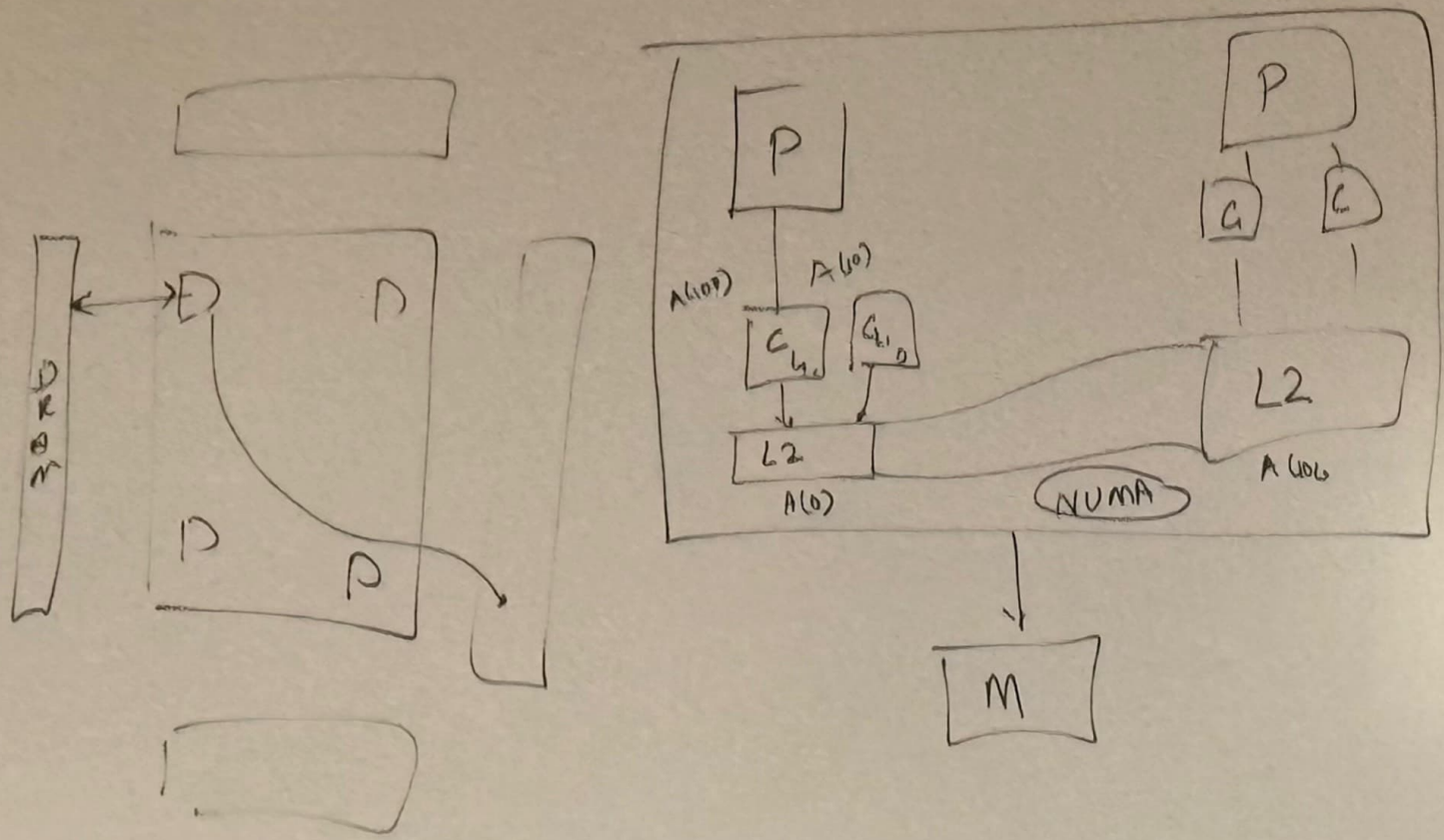
1. Organize matrix into BxB sub-matrices.
2. Iterate through blocks (jj, kk).
3. For each block, iterate through its elements (i, j).

What type of locality does this improve? Hint: Track the re-use of matrix elements during computation.



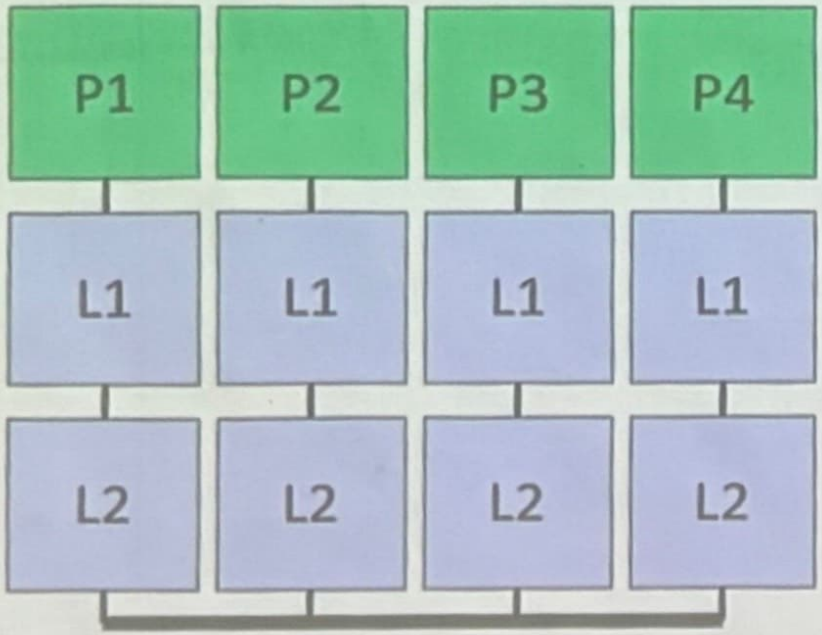


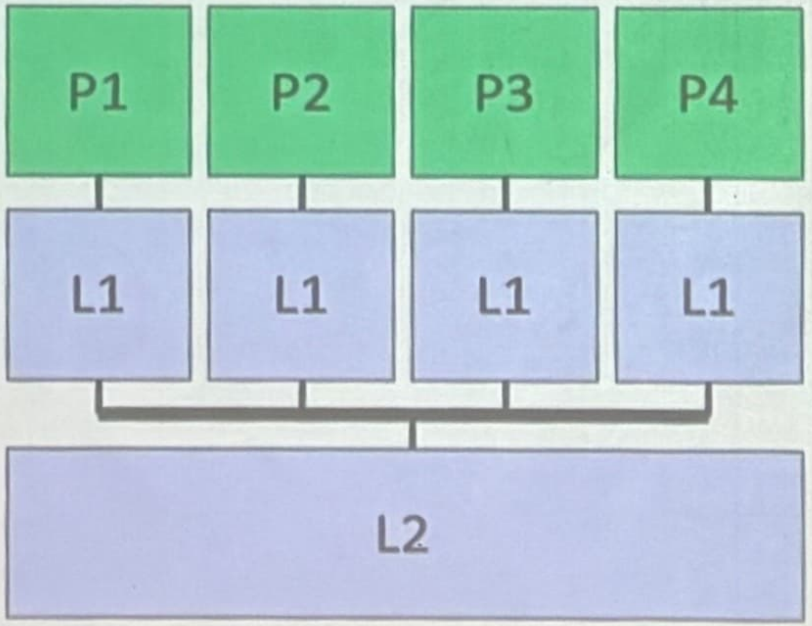
Not touched Old access New access


P
P
P
P
P
L2
L2
NUMA
M

Shared Vs. Private Caches in Multi-Core

- What are the pros/cons of a shared L2 cache?





Shared Vs. Private Caches in Multi-Core

- Advantages of a shared cache:
  - Space is dynamically allocated among cores
  - No waste of space because of replication
  - Potentially faster cache coherence (and easier to locate data on a miss)

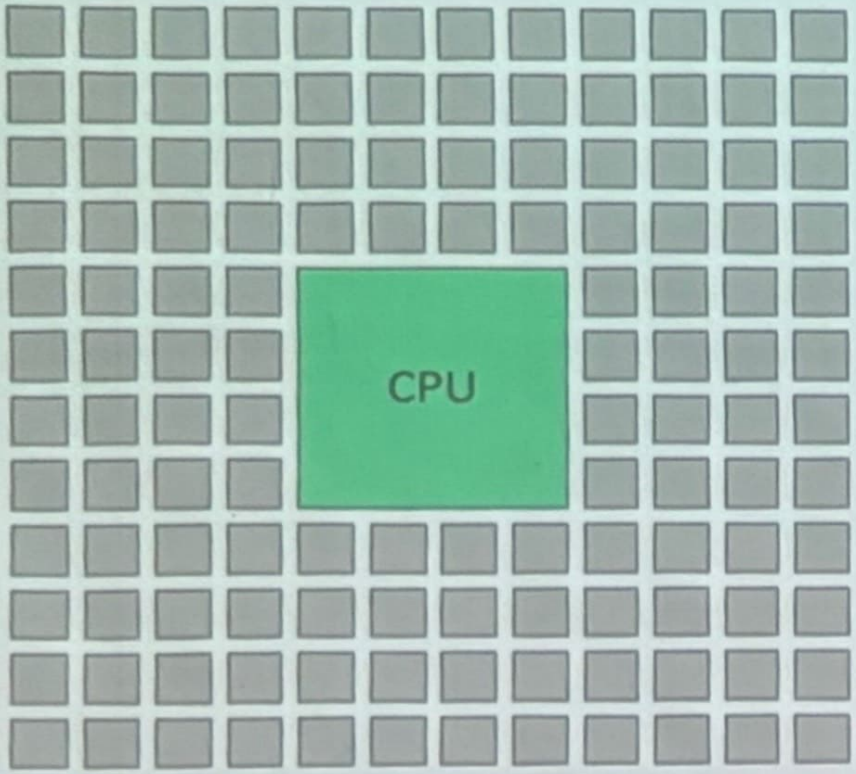
- Advantages of a private cache:
  - small L2 → faster access time
  - private bus to L2 → less contention

UCA and NUCA

- The small-sized caches so far have all been uniform cache access: the latency for any access is a constant, no matter where data is found

- For a large multi-megabyte cache, it is expensive to limit access time by the worst case delay: hence, non-uniform cache architecture

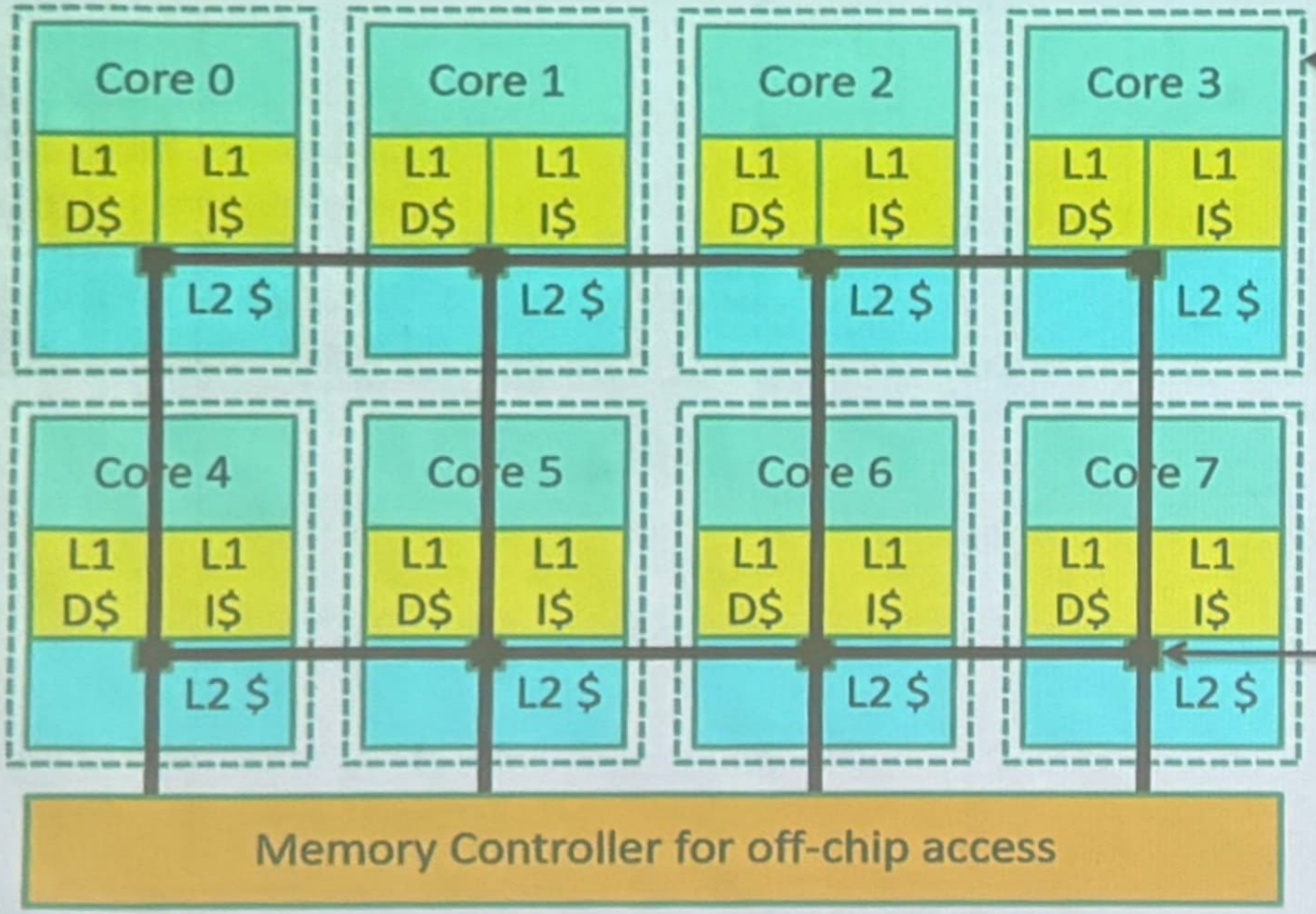
Large NUCA



Issues to be addressed for Non-Uniform Cache Access:

- Mapping
- Migration
- Search
- Replication

Shared NUCA Cache



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

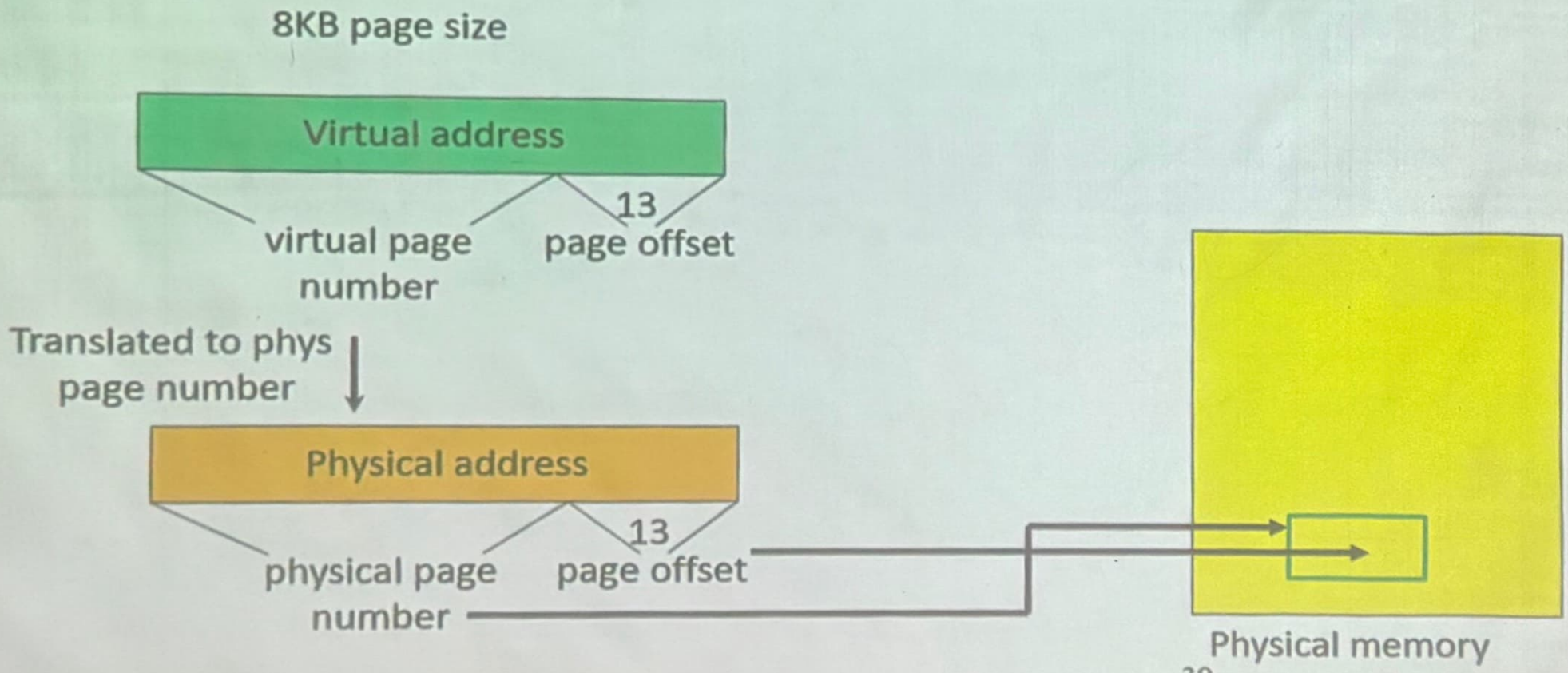
The cache controller forwards address requests to the appropriate L2 bank and handles coherence operations

Virtual Memory

- Processes deal with virtual memory – they have the illusion that a very large address space is available to them
- There is only a limited amount of physical memory that is shared by all processes – a process places part of its virtual memory in this physical memory and the rest is stored on disk
- Thanks to locality, disk access is likely to be uncommon
- The hardware ensures that one process cannot access the memory of a different process

Address Translation

- The virtual and physical memory are broken up into pages



Problem 1

- Assume a large shared LLC that is tiled and distributed on the chip. Assume 16 tiles. Assume an OS page size of 8KB. The entire LLC has a size of 32 MB, uses 64-byte blocks, and is 8-way set-associative. Which of the 40 physical address bits are used to specify the tile number? Provide an example page number that is assigned to tile 0.

Memory Hierarchy Properties

- A virtual memory page can be placed anywhere in physical memory (fully-associative)
- Replacement is usually LRU (since the miss penalty is huge, we can invest some effort to minimize misses)
- A page table (indexed by virtual page number) is used for translating virtual to physical page number
- The memory-disk hierarchy can be either inclusive or exclusive and the write policy is writeback

HPC 18 Sept

We looked at processor to reduce Execution time
but we get Data from memory, slower from processor
So we try getting something almost as fast as processor

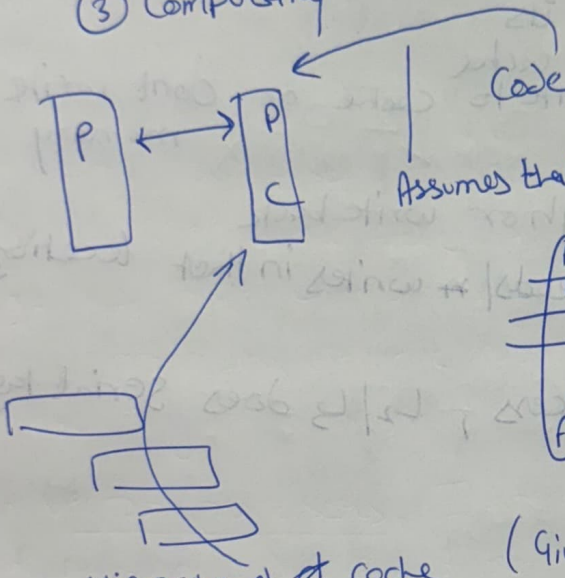
Review Cache things

How things behave in context of pipeline, out of order, speculation etc.

Types of Cache Miss

Q Can fully Associative cache have more misses than a direct mapped cache of same size?

1. Capacity
2. Conflict
3. Compulsary

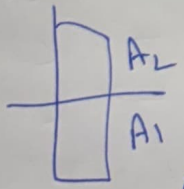


(Gives Latency problem when we bring data from far)

Do you start with Code cache at every switch?
(It starts with warm cache or cold cache?)

$A_1 \rightarrow A_2 \rightarrow A_1$ — will it get compulsary miss

Context switch



Cache (2 way set Associative)
Presumably can hold A_1, A_2 relevant code without needing any possible eviction

} Direct map cache then def. eviction needed

32-Arch
-X-86

Basics of Cache [Instruction cache is Read only, Data Cache is Read and write]

L_1 Cache split into L_2 and L_3

L_1/L_2 can be inclusive exclusive and non-inclusive

↑ out of sync

On a write we can do write allocate / no allocate
(Must update memory, cache is irrelevent)

Data path can go via cache or seperately

→ processor's computation over as its written in cache

make relation valid first then write to cache or dont write to memory

- every update is write through or write back
(improving further reads/writes in that location)

L_1 does parallel tag data access, L_2/L_3 does serial tag/Data access

Key equation

Avg memory Access time

[Hit time] + [Miss prob] * [Miss Penalty]

larger memory = larger Hit time

Smaller cache - larger miss rate

(Points from Sides copy)

Access to data (memory to cache must be reduced

$L_1 \rightarrow L_2 \rightarrow L_3 \rightarrow$ Dram

Serial vs parallel access (Slide) copy

Cache miss causes

Compulsary → Always happen when prog start

Capacity / conflict → can be reduced
- → Handle with Associativity
- → Handle with Replacement policy and size

we did out of order & Speculation
- ↳ branch

to reduce stall cycles

Does it have any impact on AMAT?
intuitively think, we are doing branch pred.
and we are assuming taken
so ~~br~~ if it is not in cache, if it mis pred
there will be stall penalty and if correctly
pred ~~e~~ no need to Access Memory / stall
avoid

Same thing can be said for out of order
↳ if Doing Dynamically?

→ we did speculation with branch pred
we could speculate what to speculate in memory or cache

Speculation
- ↳ branch pred
- ↳ pre fetching

Tolerating Miss Penalty (slide)
↳ can do some useful work
non blocking cache

ld R0, 0(R1)
R1 ← address
↓ Miss
Cache
↓
memory

Cache performance (Slide)

Reducing Miss Rate

key thing Size goes down Higher associativity
its like have bucket 16, now grouping them in 2
we can address/store data in 8 of them

Any replacement policy, there might be usecases where we thrown out a line and we need it

2 loc { odd addresses / even addresses

Maybe accessing Alternatively?

or Access pattern is → 0 1 1 2 0 3 1
x x ✓ x x x x

Technique to reduce cache Modern processor, miss
Do not throw it away Completely
We throw it into another Cache that Contains thrown out blocks

x - Miss

(location you thrown out was accessed soon after thrown out)

1 location (Direct map cache)

0 2 0 2 0 (Always miss)
x x x x x

So we maintain list of Addresses thrown out in a victim cache (Slide)

Victim cache (Slide)

Replacement policies
- Psudo LRU
- NRU

Actual LRU has to maintain time Stramp which is too expensive

take a bit for ~~each~~

So everytime that location is touched you put a 1

we did someting similar in branch pred

Think?

At each time you access that location you toggle it, eachtime you use location you make it 1 or 0

Cache write Policy Alternatives (Slide)

Reducing write time

Write must first check if its available in cache
if available write
else write to memory with write allocate or No allocate
two miss can happen one to memory one to cache
way ~~to~~ do it is use write buffer (like a queue)
Put it in seperate buffer so you are not stalling or right process has finished

- Cache ~~Hire~~ Hierarchy. (Slide)
- Typical L_1 L_2 Interactions (Slide)
- Prefetching (Slide)
  → employed for any of cache levels
  → binging anticipated data in cache
  It can cause Cache pollution

Q- Cloud? → A collection of servers, that we ~~go~~ give out with some specifications, 32 G.B of memory with some cores
1-1 mapping of logical and physical cors

4 tenents – 4 logical processor – 1 physical core

Challenges is pre-fetch (slide)

Hardware (Slide)
Prefetching – (not new)
L counciously / uncounciously using

A_3 A_4 A_5 A_6
↑ miss — prefetches

locality (Hardware was doing)

Same thing Compiler can also do.

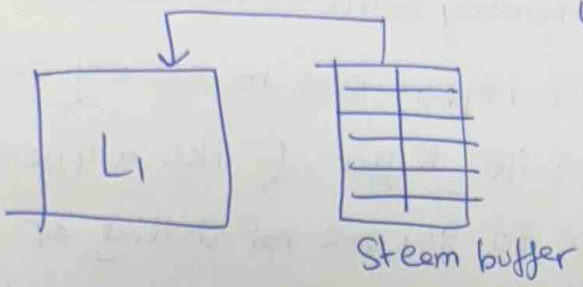
Hardware prefetch (Slide)
L one block look ahead
L strided prefetch

Software prefetch
↳ Prefetch P locations

Steam Buffer (slide)

↳ where you keep data as its coming in

Memory Single ported



when No cache miss then take data from buffer bring to cache

Cache and code (Slide)

→ Restructuring
→ kill cache pollution
→

Loop inter change

if Data is row major format, accessing in column major order, then change in the order

Loop fusion (slide)

Matrix Multiply (slide) – (X=YZ) Naïve

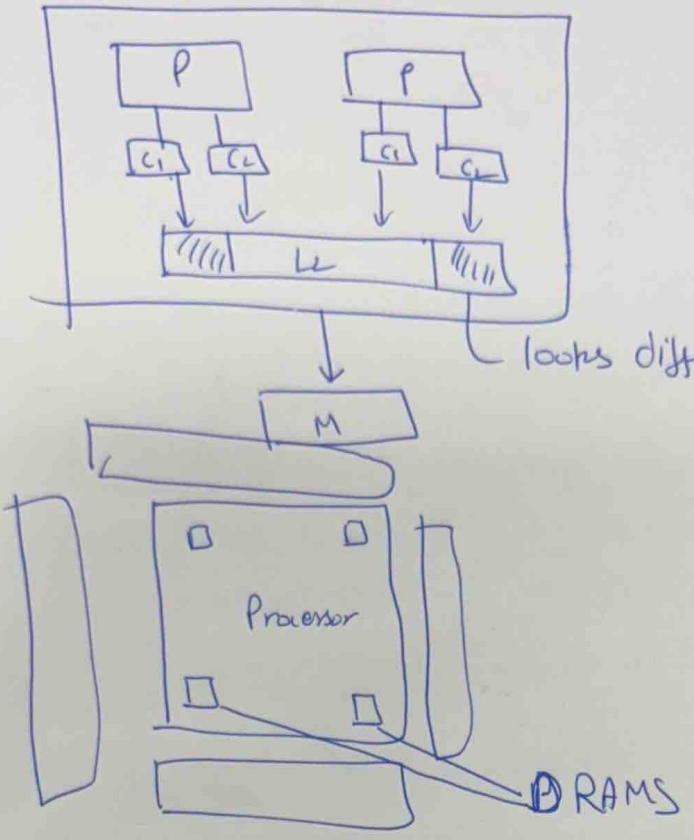
entire matrix does nt fit
one row / major doesnt fit

do in parts of a, b's

(Given more in book Paterson, look from Cache chapter)

(Review of whats expected in basics)

we talk How can multiple processor sharing cache maintain consistency?



Compiler need to be very careful in partitioning

Shared vs private cache in Multicore (Slide)

Shared L2 — Larger cache — Contengent
(Adv) (disadv)

UCA vs NUCA Slide

Virtual Memory — what is virtual about virtual memory?
└ Addresses
is it inf?
└ NO, limited by size of Reg
R0 (content of some Address)

(Good Question)

(Slide)

How hardware ensure a programs memory
Can not be accessed by another?

→ Problems (slide)

→ Memory Hierarchy properties (Slide)

typically write policy is write back