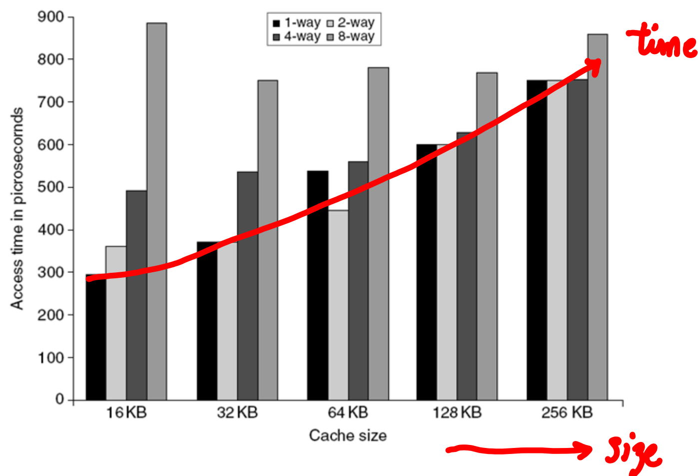
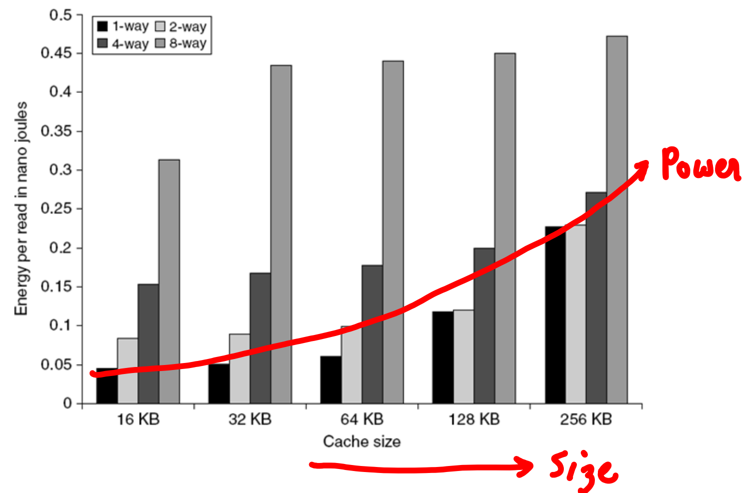
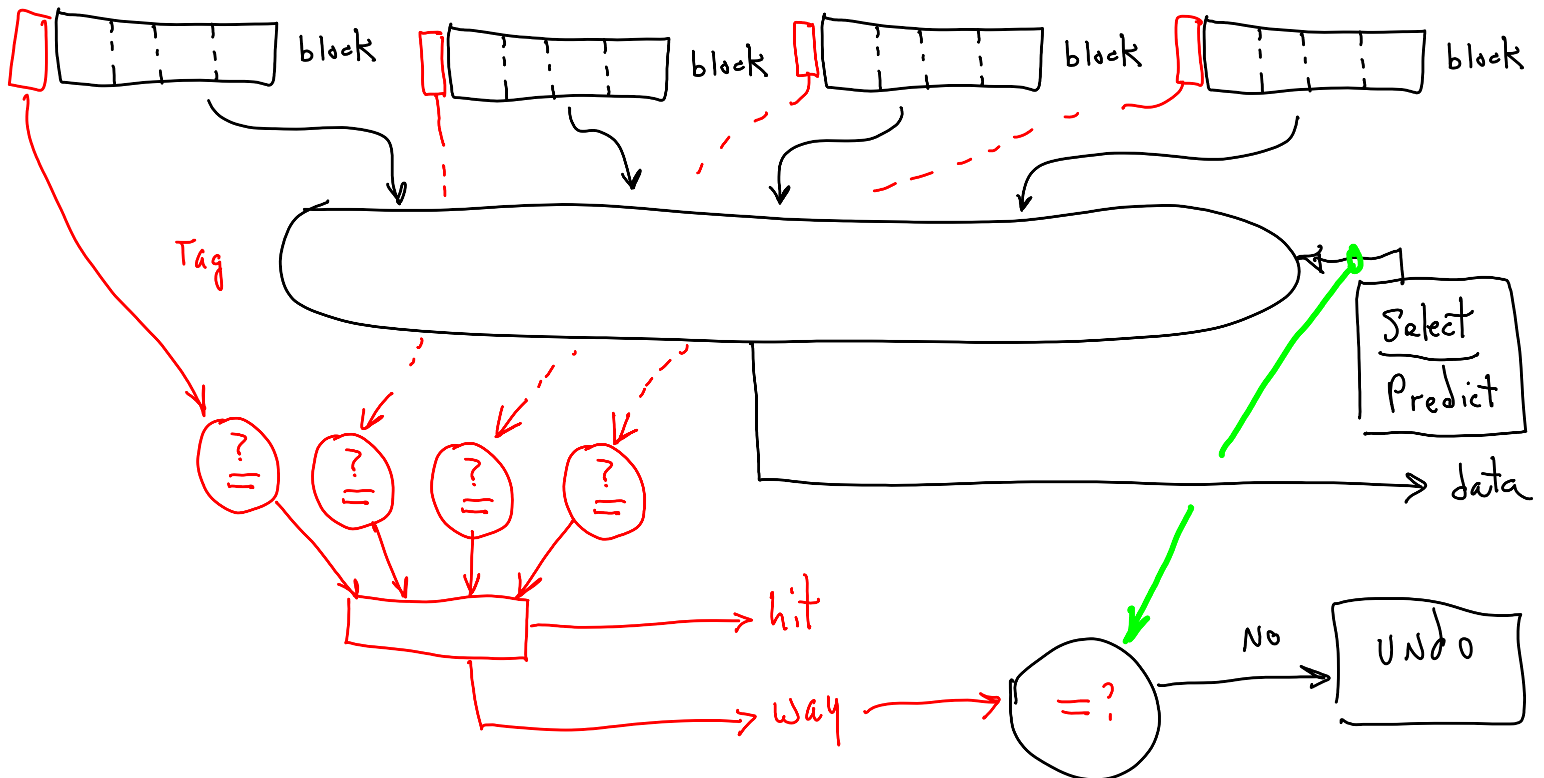
Cache Optimizations

- Small and simple first level caches
  - Critical timing path:
    - addressing tag memory, then
    - comparing tags, then
    - selecting correct set
  - Direct-mapped caches can overlap tag compare and transmission of data
  - Lower associativity reduces power because fewer cache lines are accessed



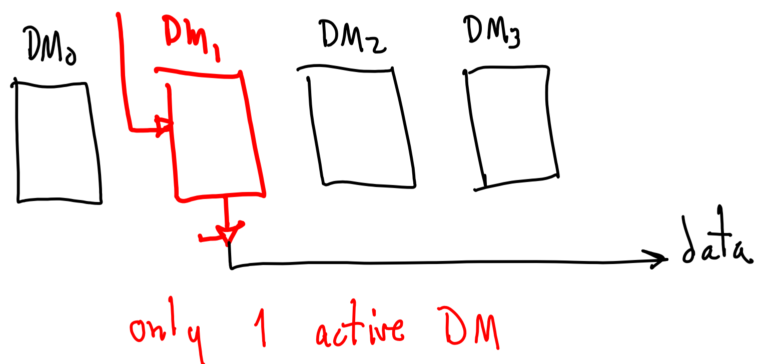


- To improve hit time, predict the way to pre-set mux
  - Mis-prediction gives longer hit time
  - Prediction accuracy
    - > 90% for two-way
    - > 80% for four-way
    - I-cache has better accuracy than D-cache
  - First used on MIPS R10000 in mid-90s
  - Used on ARM Cortex-A8
- Extend to predict block as well
  - "Way selection"
  - Increases mis-prediction penalty



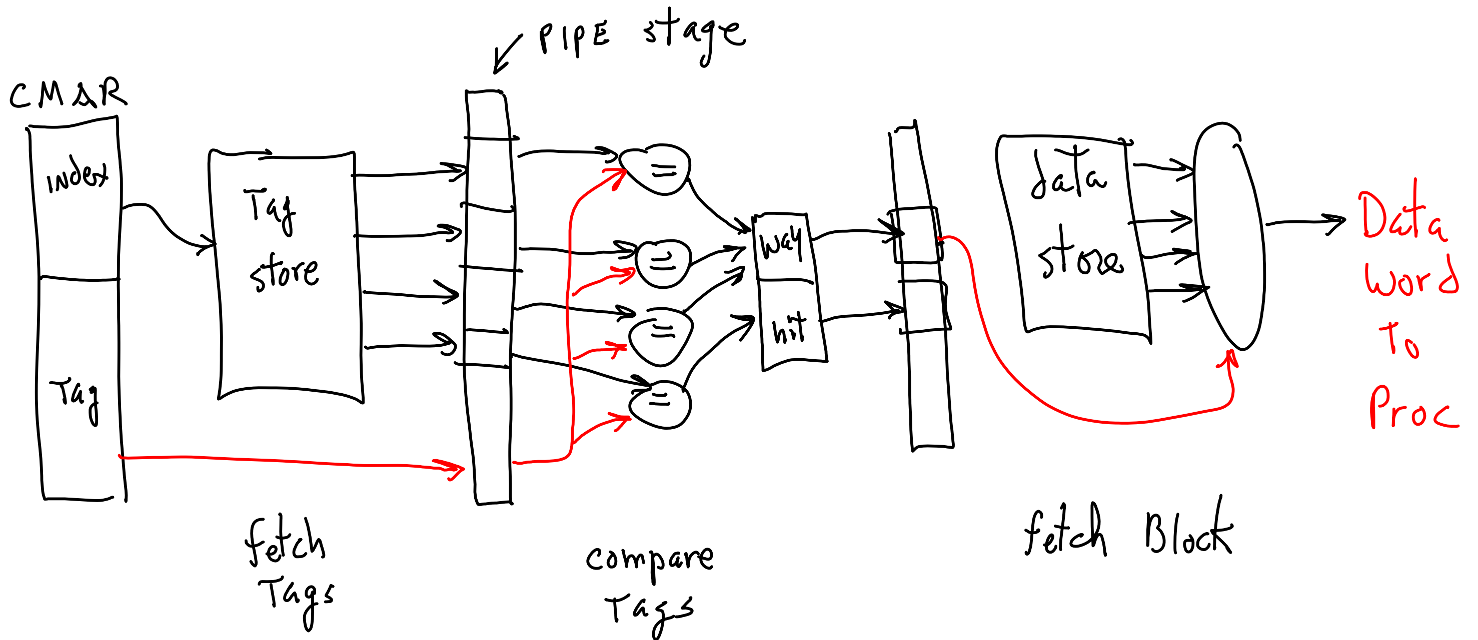
Way Select

Ignore other ways ==> DM



- Pipeline cache access to improve bandwidth
  - Examples:
    - Pentium: 1 cycle
    - Pentium Pro – Pentium III: 2 cycles
    - Pentium 4 – Core i7: 4 cycles

- Increases branch mis-prediction penalty
- Makes it easier to increase associativity



1 word data to processor per cycle

pipeline fill/drain overhead

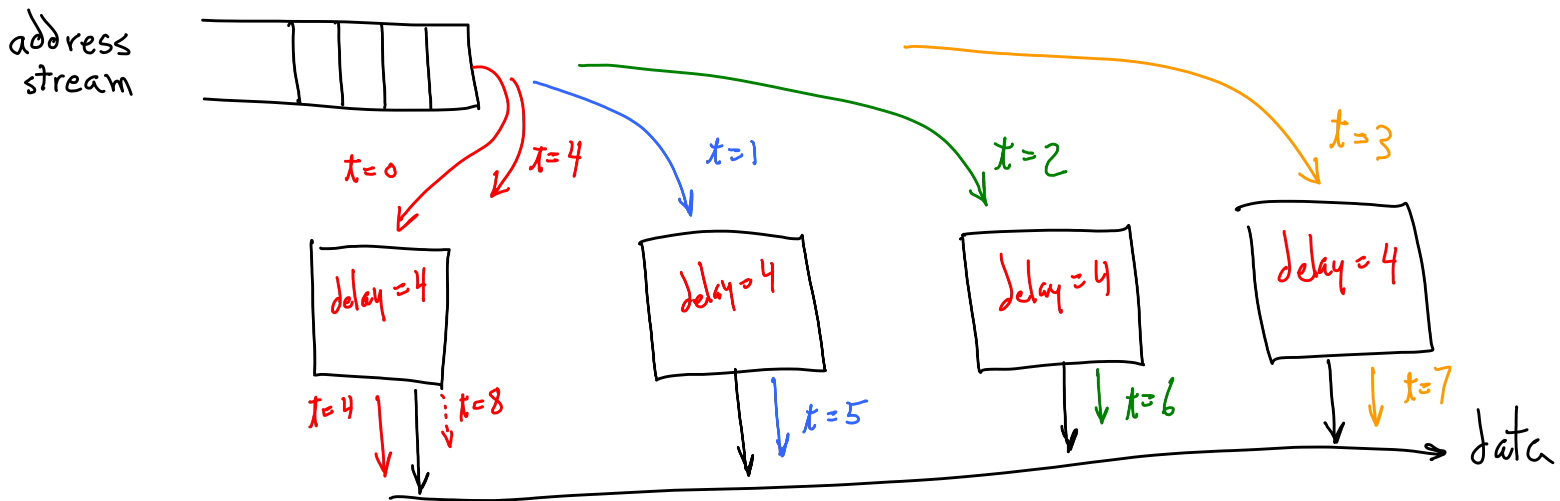
pipeline hazard overhead

- Organize cache as independent banks to support simultaneous access
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2

- Interleave banks according to block address

Block address	Bank 0	Block address	Bank 1	Block address	Bank 2	Block address	Bank 3
0		1		2		3	
4		5		6		7	
8		9		10		11	
12		13		14		15	

Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

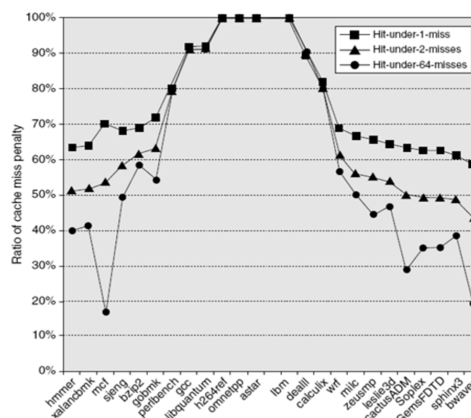


General Principle: Keep working to hide latency

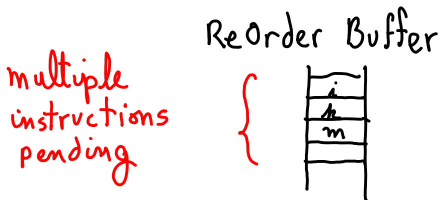
--- Can we find other work to do?
--- Do we have a mechanism for that?
--- Multiple threads (switching), out-or-order execution, loop unrolling

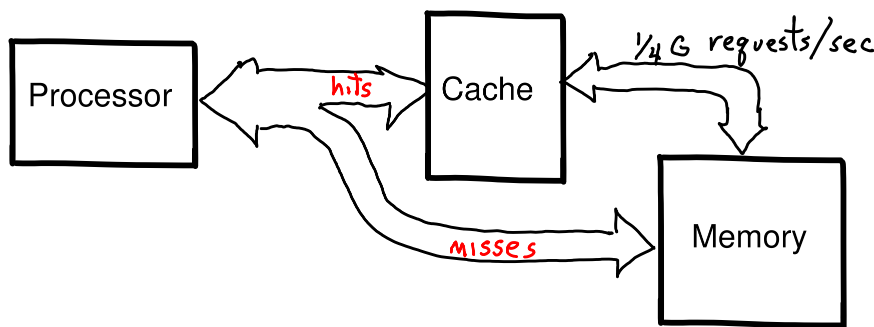
Nonblocking Caches

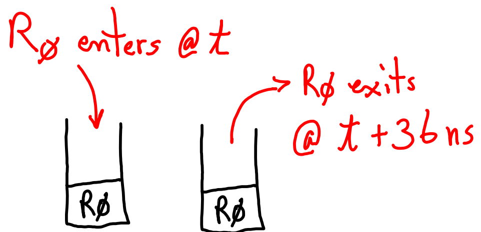
- Allow hits before previous misses complete
  - "Hit under miss"
  - "Hit under multiple miss"
- L2 must support this
- In general, processors can hide L1 miss penalty but not L2 miss penalty

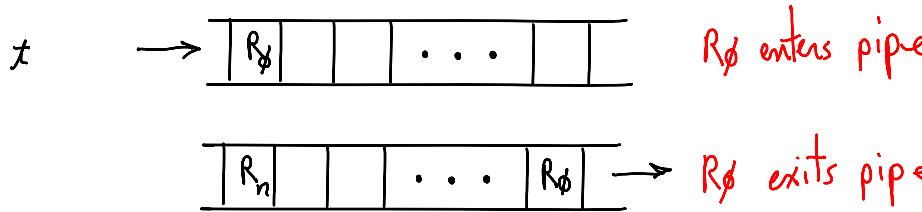


E.G. 36 ns latency
16 GB/sec Memory Bandwith } → $\frac{1}{4}$ G requests/sec → $\frac{4\text{ ns}}{\text{request}}$
64-B cache block per request }







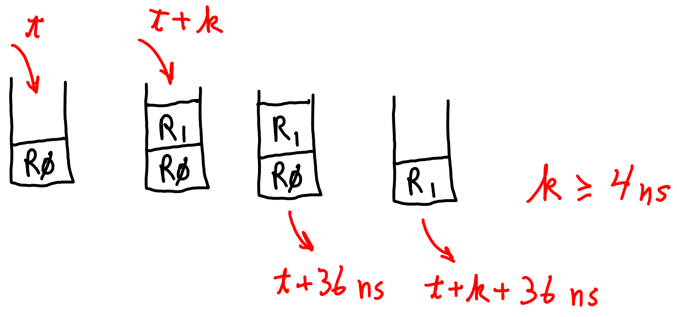


One concurrent miss supported

Mem bandwidth used:
64B/36ns ~ 2 GB/sec

(1/8)

How many, at most, concurrent requests can we handle? Do we need to handle if we want to use all our bandwidth?



Two concurrent misses supported

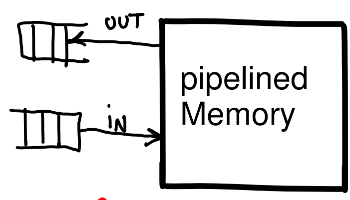
Bandwidth = 2(64B)/(k + 36)ns
< 2(64/40) GB/sec
~ 3 GB/sec

(~1/5)

Maximum concurrent misses we can support?

at steady state, maximum request rate we can fulfill:

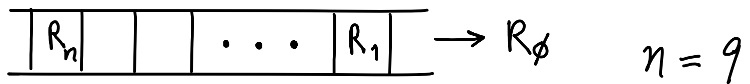
required bandwidth = (1 req/4ns)(64B/req) = 16 GB/sec



As long as queues are steady, n → ∞!

If requests take exactly 36 ns latency?

(36 ns latency / req) = n (steps down pipeline) (4ns / step)

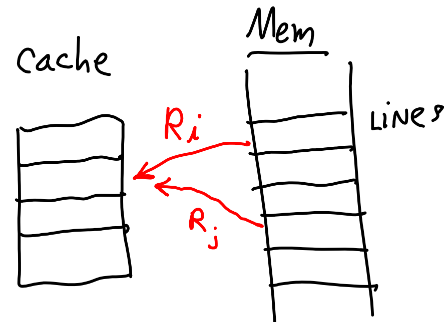


$n = 9$

Suppose

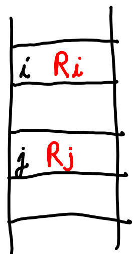
Collision (different lines)

R_i reads in L_i } to same cache
R_j reads in L_j } location



References R_i and R_j Collide

Reorder Buffer

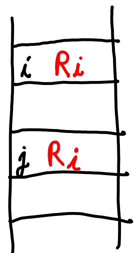


⇒ R_i when R_i returns i executes

⇒ R_j when R_j returns j executes (replaces R_i w/ R_j)

Collision (same line)

Reorder Buffer

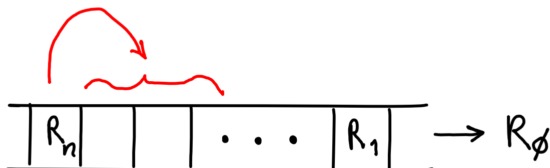


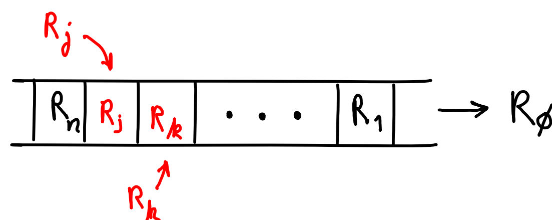
⇒ R_i

⇒ nothing (request already pending)

2 pending references, but only 1 request to memory

Suppose: 50% chance of collision w/ prior requests





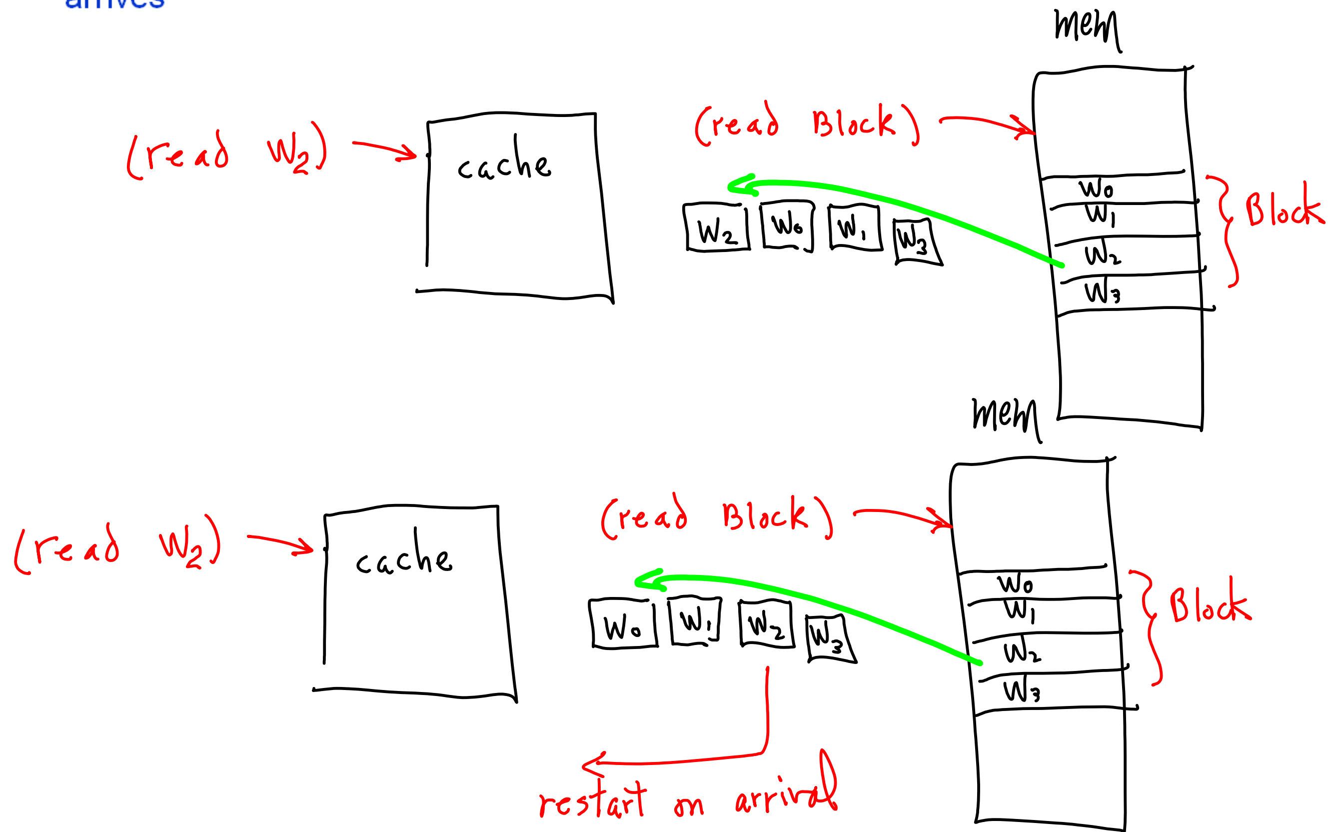
for every request in process 1 is piggybacked

IF
--- We want latency = min = 36ns
--- We want to utilize full memory bandwidth

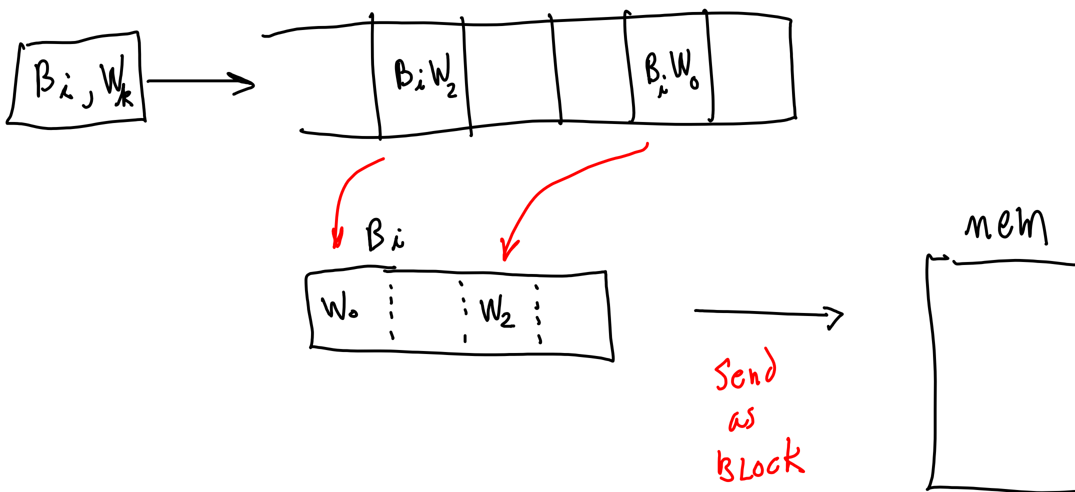
We have to be able to handle 18 concurrent misses.

We need a mechanism that can still find other work to do, even though 18 instructions are queued waiting for data.

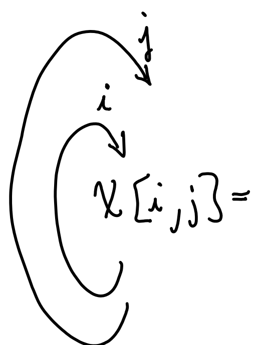
- Critical word first
  - Request missed word from memory first
  - Send it to the processor as soon as it arrives
- Early restart
  - Request words in normal order
  - Send missed work to the processor as soon as it arrives



- When storing to a block that is already pending in the write buffer, update write buffer
- Reduces stalls due to full write buffer
- Do not apply to I/O addresses



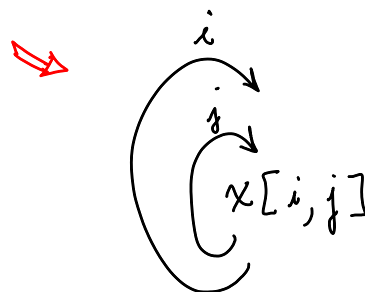
- Loop Interchange
  - Swap nested loops to access memory in sequential order



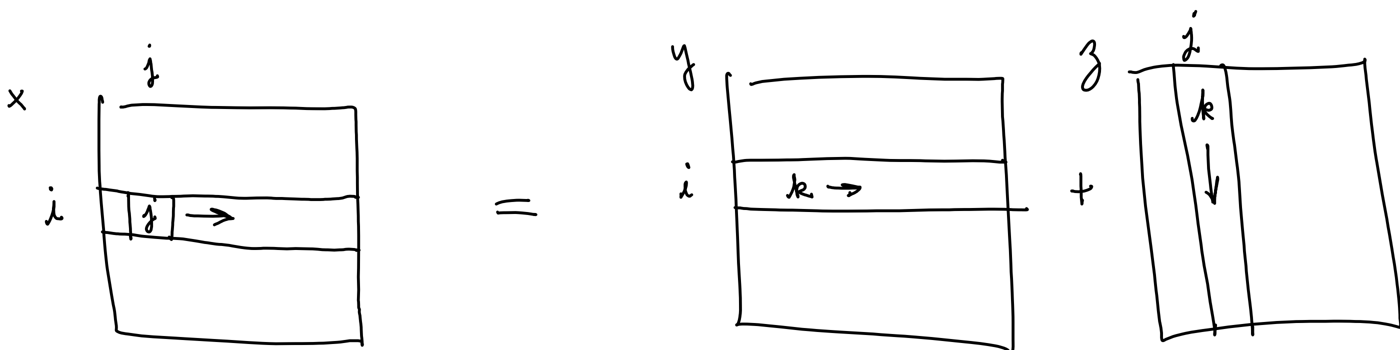
x[0, 0] =
x[1, 0] =
x[2, 0] =
⋮
x[n, k] =

x[0,0] =
x[0,1] =
x[0,2] =
⋮

re-order
To match
Blocks in mem

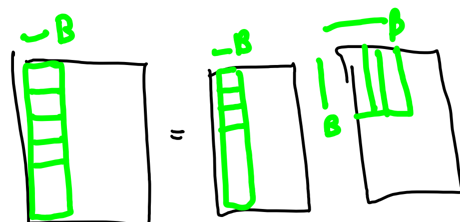


- Blocking
  - Instead of accessing entire rows or columns, subdivide matrices into blocks
  - Requires more memory accesses but improves locality of accesses



i j k x[i, j] = y[i, k] + z[k, j]

kk + B
jj + B

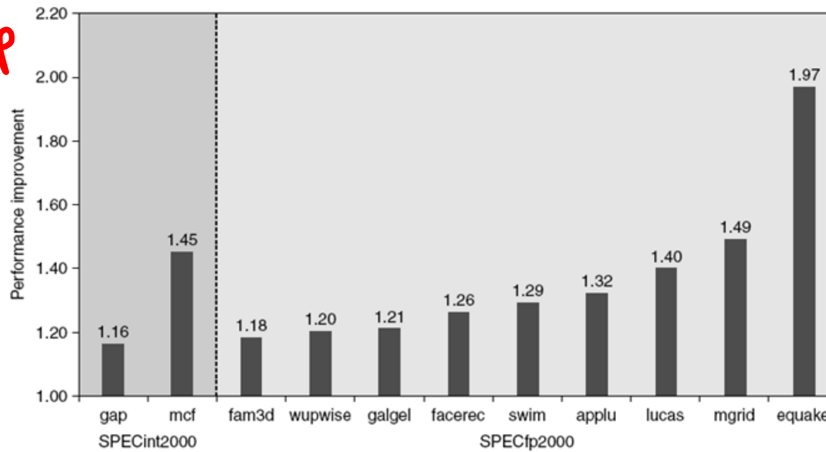


- Fetch two blocks on miss (include next sequential block)

Hardware Prefetch

→ no exceptions (faults)!

Speedup

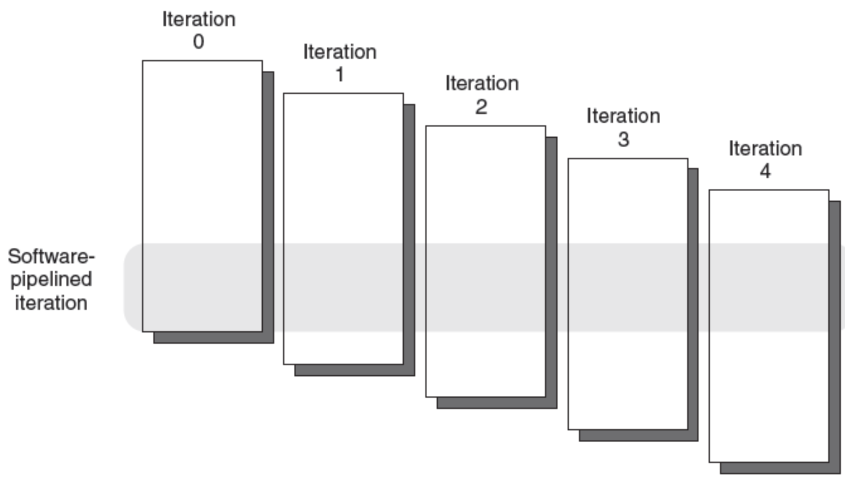


Pentium 4 Pre-fetching

- Insert prefetch instructions before data is needed
- Non-faulting: prefetch doesn't cause exceptions

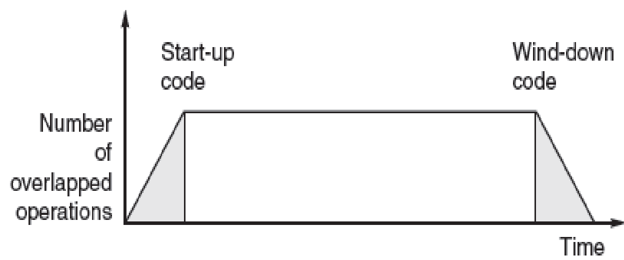
- Register prefetch
  - Loads data into register
- Cache prefetch
  - Loads data into cache

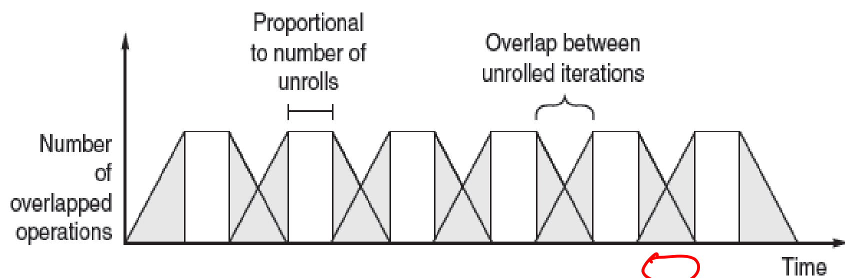
- Combine with loop unrolling and software pipelining

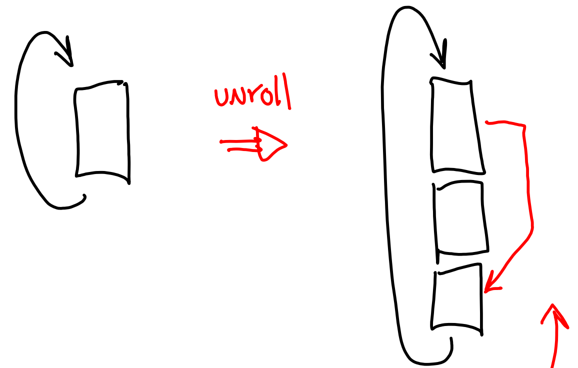


find independent (data)
instructions
schedule together

Figure H.1 A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop. The start-up and finish-up code will correspond to the portions above and below the software-pipelined iteration.







dependencies → serialization