

332

Advanced Computer Architecture

Chapter 2: Caches and Memory Systems

January 2018

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3rd, 4th, 5th and 6th eds), and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

Average memory access time:

$$\text{AMAT} = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

There are three ways to improve AMAT:

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache

We now look at each of these in turn...

Reducing Misses

● Classifying Misses: 3 Cs

● **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. Also called **cold start misses** or **first reference misses**.

(Misses in even an Infinite Cache)

● **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, **capacity misses** will occur due to blocks being discarded and later retrieved.

(Misses in Fully Associative Size X Cache)

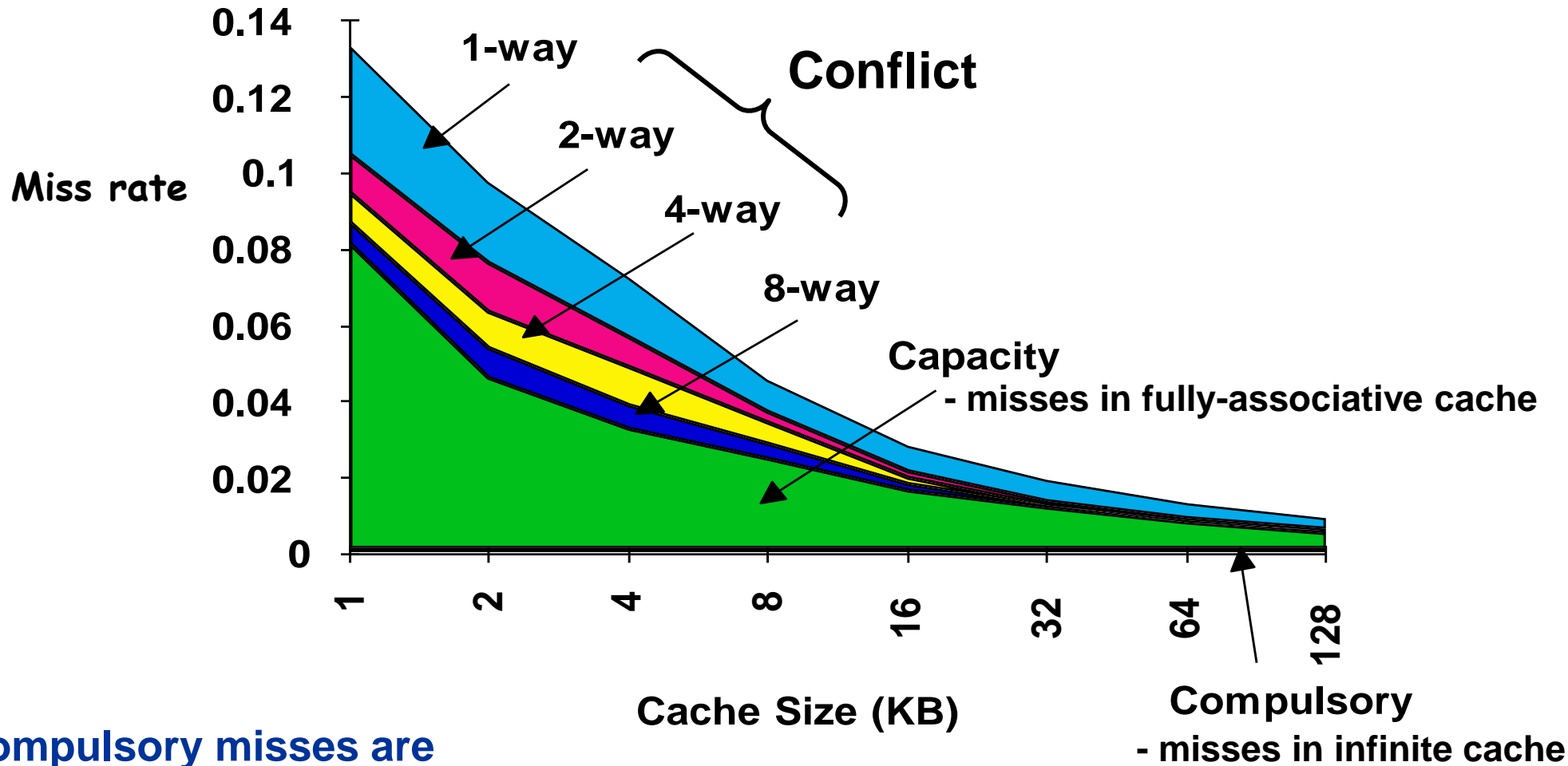
● **Conflict**—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called **collision misses** or **interference misses**.

(Misses in N-way Associative, Size X Cache)

● Maybe four: 4th “C”:

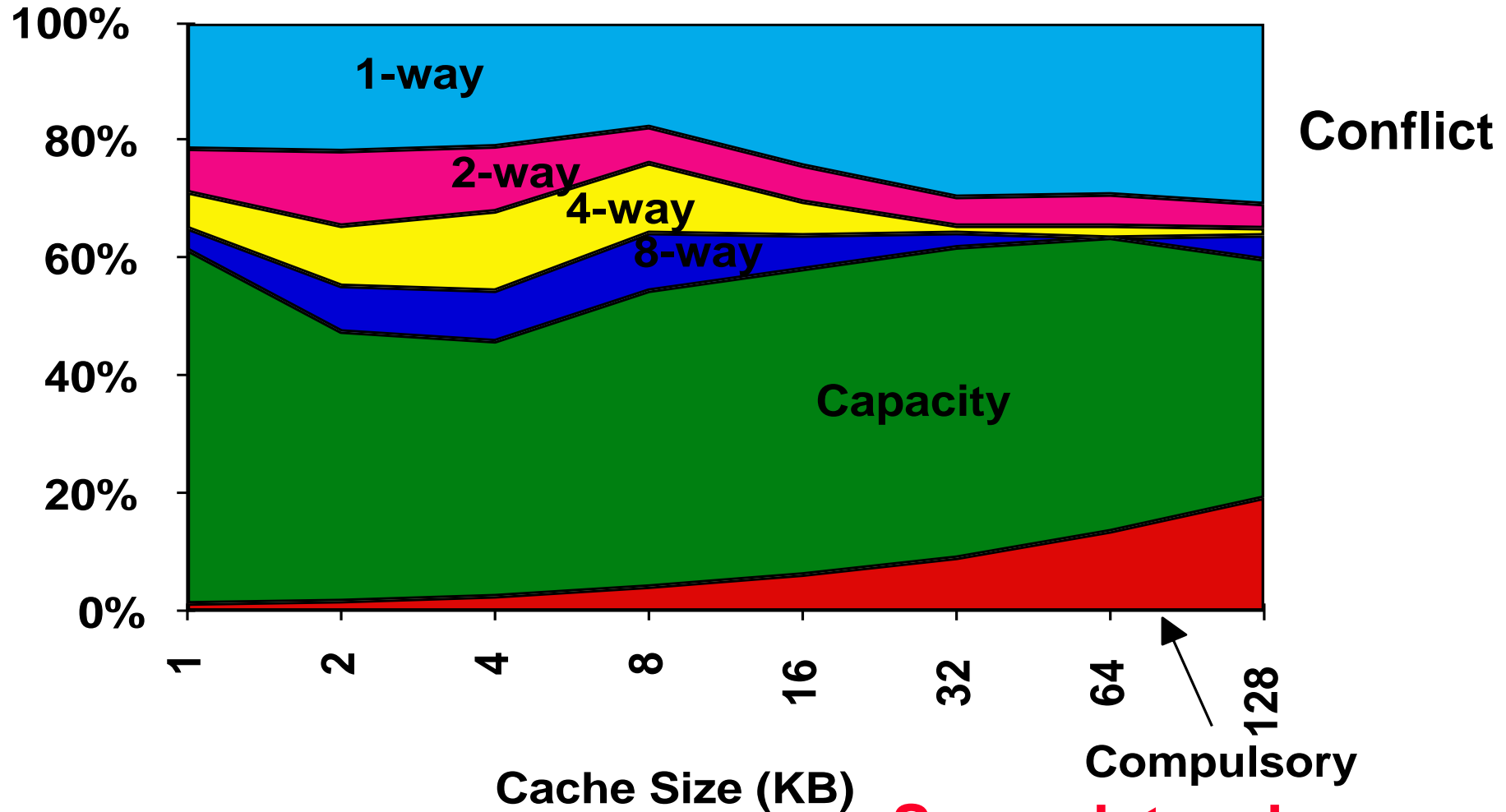
● **Coherence** - Misses caused by cache coherence: data may have been invalidated by another processor or I/O device

3Cs Absolute Miss Rate (SPEC92)



Compulsory misses are
often vanishingly
Few (unless??)

3Cs Relative Miss Rate



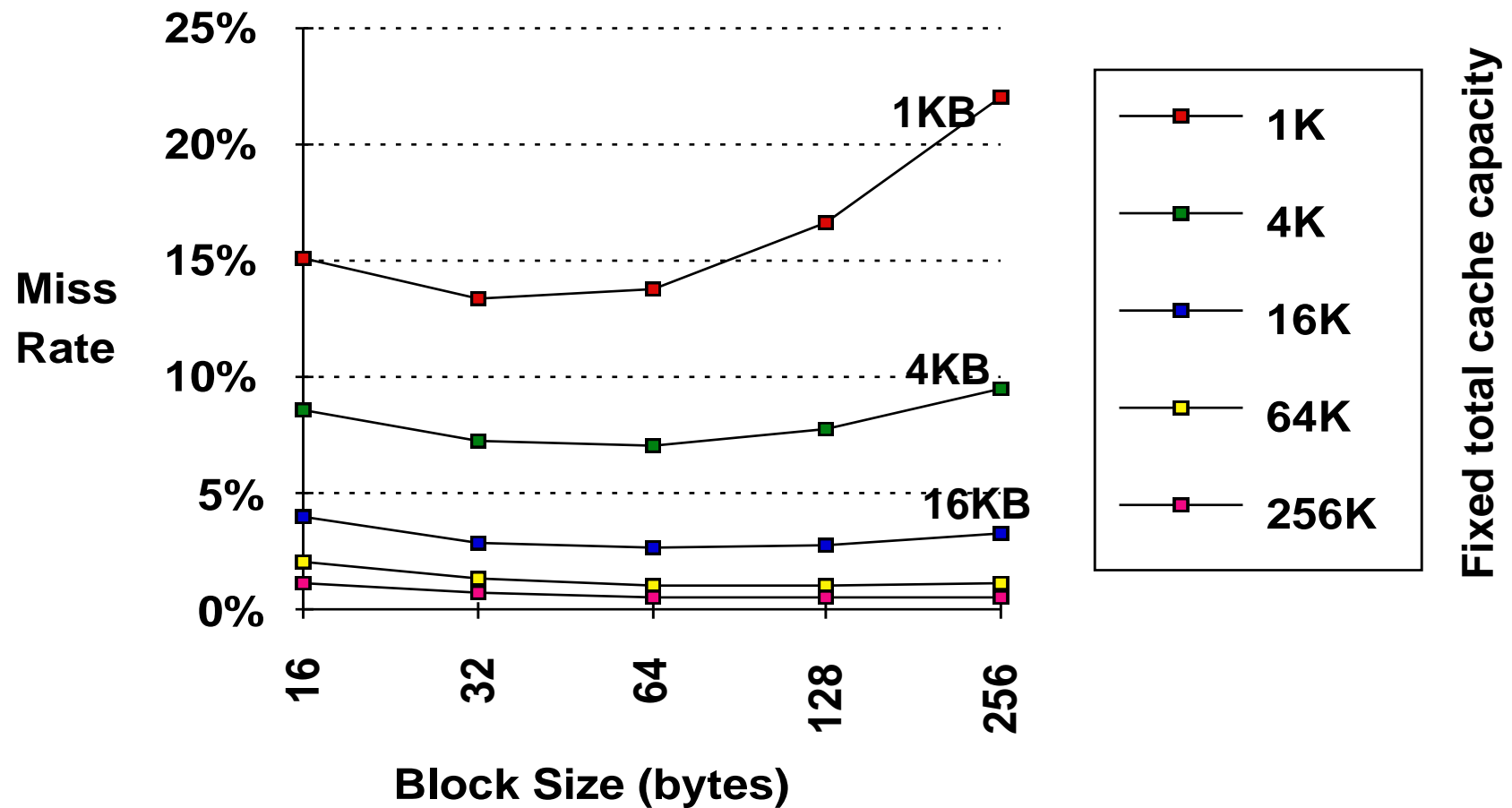
Same data, shown as
proportion of total

How We Can Reduce Misses?

- **3 Cs: Compulsory, Capacity, Conflict**
- **In all cases, assume total cache size not changed:**
- **What happens if:**
 - 1) **Change Block Size:**
Which of 3Cs is obviously affected?
 - 2) **Change Associativity:**
Which of 3Cs is obviously affected?
 - 3) **Change Compiler:**
Which of 3Cs is obviously affected?

We will look at each of these in turn...

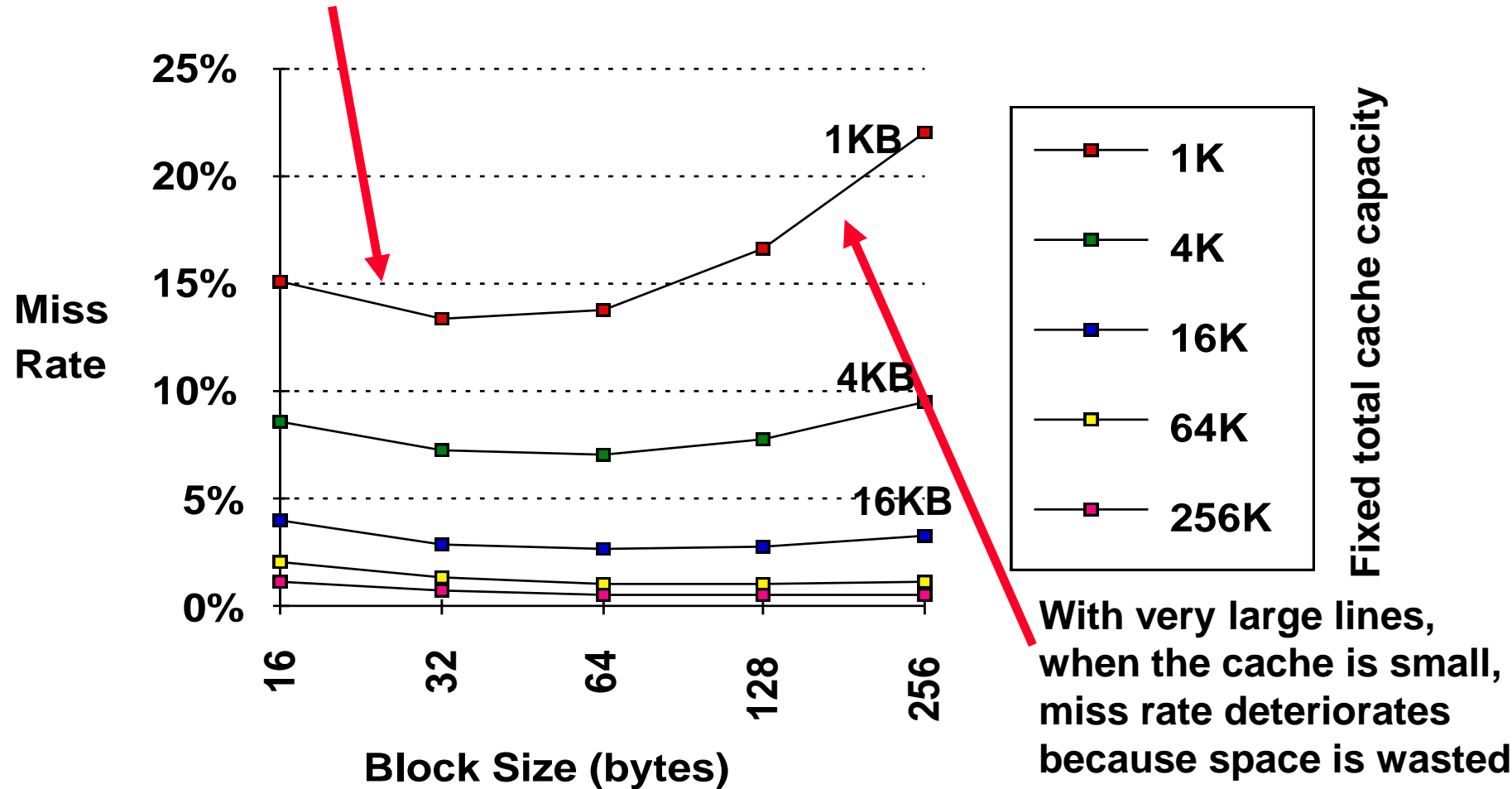
1. Reduce Misses via Larger Block Size



Bigger blocks allow us to exploit more spatial locality – but...

1. Reduce Misses via Larger Block Size

Initially miss rate is improved due to spatial locality



Note that we are looking *only* at miss rate – large blocks will take longer to load (ie a higher *miss penalty*)

Later we will see

- Better ways to exploit spatial locality, such as prefetching
- Ways to reduce the miss penalty, eg critical word first and sectoring

2: Associativity: Average Memory Access Time vs. Miss Rate

● Beware: Execution time is all that really matters

- Will Clock Cycle time increase?

● Example: suppose clock cycle time (CCT) =

- 1.10 for 2-way,

- 1.12 for 4-way,

- 1.14 for 8-way

- vs. CCT = 1.0 for direct mapped

● Although miss rate is improved by increasing associativity, the cache hit time is increased slightly

Cache Size (KB)	Associativity			
	1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

- Illustrative benchmark study.
Real clock cycle cost likely smaller

(Red means A.M.A.T. not improved by more associativity)

2: Associativity: Average Memory Access Time vs. Miss Rate

● Beware: Execution time is all that really matters

- Will Clock Cycle time increase?

● Example: suppose clock cycle time (CCT) =

- 1.10 for 2-way,
- 1.12 for 4-way,
- 1.14 for 8-way
- vs. CCT = 1.0 for direct mapped

● Although miss rate is improved by increasing associativity, the cache hit time is increased slightly

(Red means A.M.A.T. not improved by more associativity)

● Solution?

- Way speculation
- See textbook

Cache Size (KB)	Associativity			
	1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

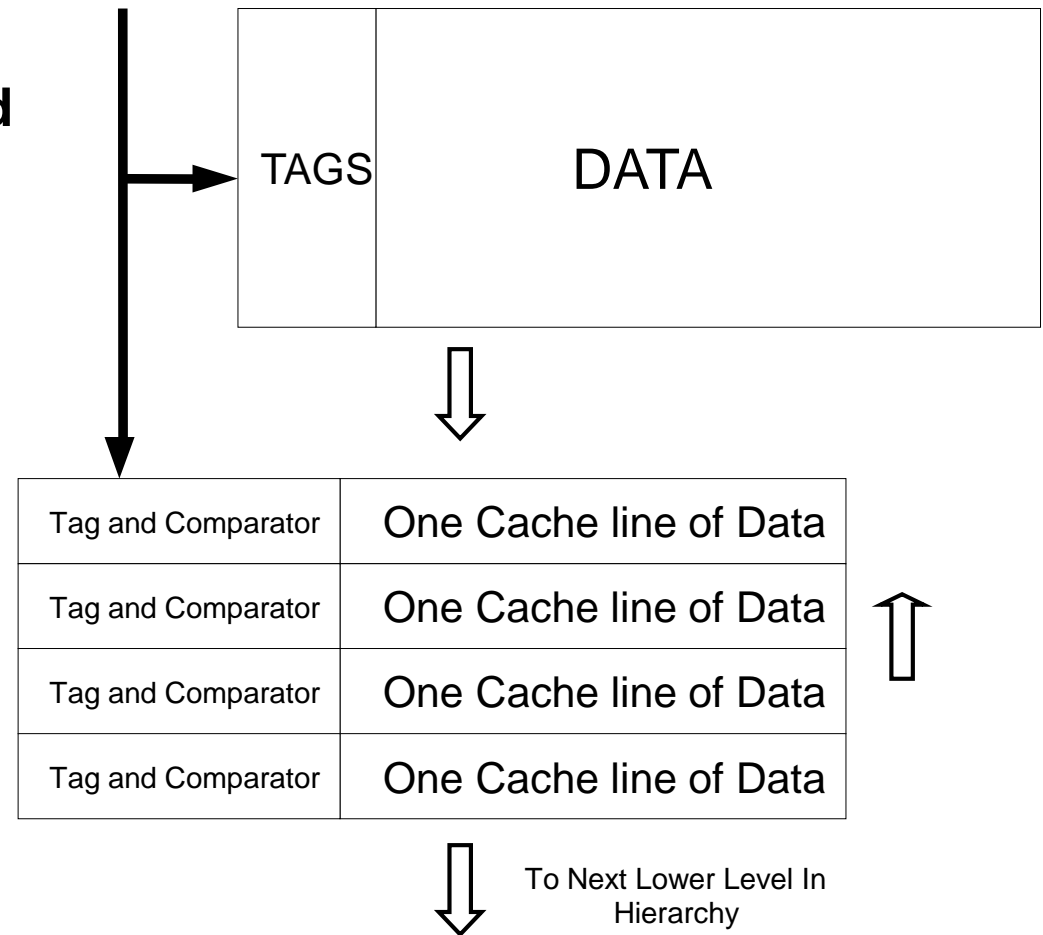
● Illustrative benchmark study. Real clock cycle cost likely smaller

Another way to reduce associativity conflict misses: “Victim Cache”

- How to combine fast hit time of direct mapped yet still avoid conflict misses?
- Add buffer to place data discarded from cache
- Jouppi [1990]: 4-entry victim cache removed 20-95% of conflicts for a 4 KB direct mapped data cache
- Used in AMD Opteron, Barcelona, Phenom, IBM Power5, Power6



HP Fellow
Director, Exascale Computing Lab
Palo Alto



(A digression: competitive algorithms

- Given two strategies

- Each strategy is good for some cases but disastrous for others
- Can we combine the two to create a good composite strategy?
- What price do we have to pay?

- Example: ski rental problem

(https://en.wikipedia.org/wiki/Ski_rental_problem)

- Example: spinlocks vs context-switching

- Example: paging (should I stay or should I go)

- Related: the Secretary problem (actually best understood as dating)

- I hope you will demand a course in competitive algorithms and apply them to diverse computer systems problems

- See <http://www14.in.tum.de/personen/albers/papers/brics.pdf>)

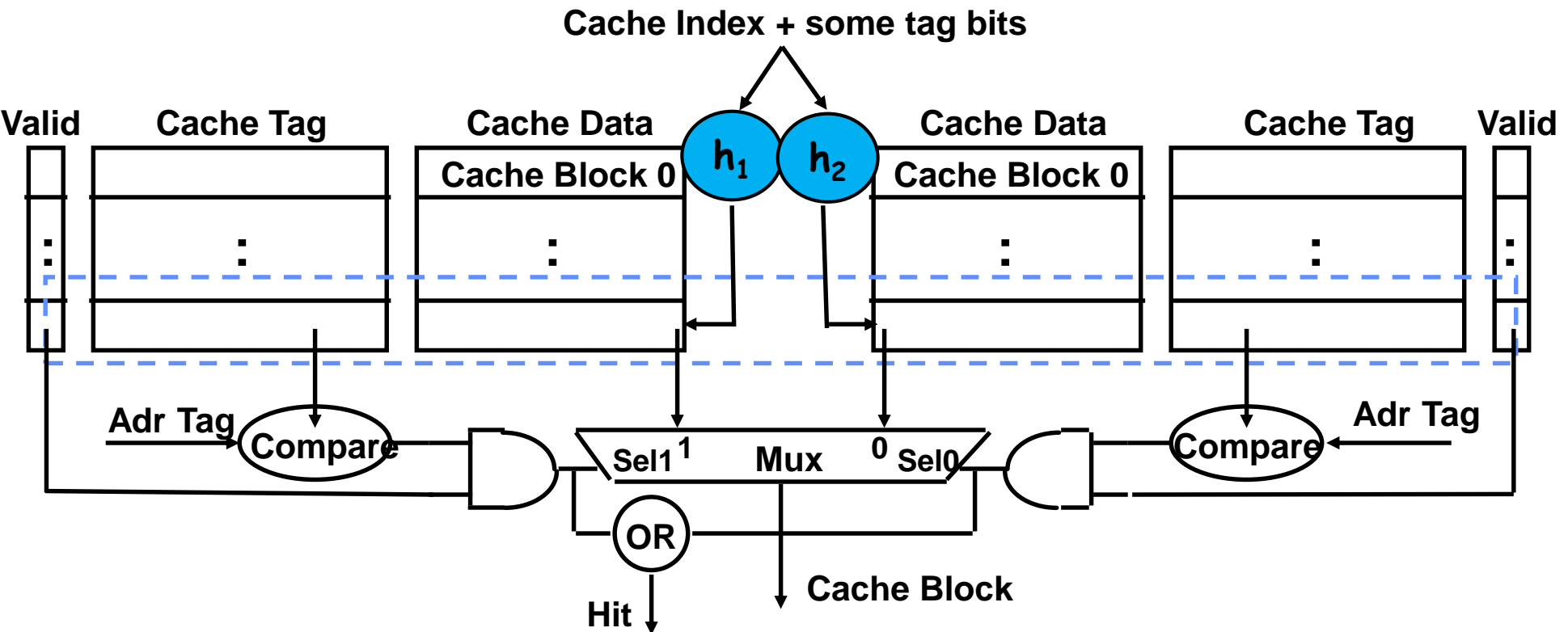
- How to timetable all of DoC and EEE's 3rd-year, 4th-year and MSc courses
- With limited number of rooms and times in the week
- There must be some clashes
- Suppose you want to take two courses, "ACA" and "BigData"
- If you're lucky they are scheduled on different slots
- If not, they clash every week!

Week 1	Mon@2	ACA	BigData
	Tue@2		
	Wed@2		
	Thu@2		
Week 2	Mon@2	ACA	BigData
	Tue@2		
	Wed@2		
	Thu@2		
Week 3	Mon@2	ACA	BigData
	Tue@2		
	Wed@2		
	Thu@2		
Week 4	Mon@2	ACA	BigData
	Tue@2		
	Wed@2		
	Thu@2		

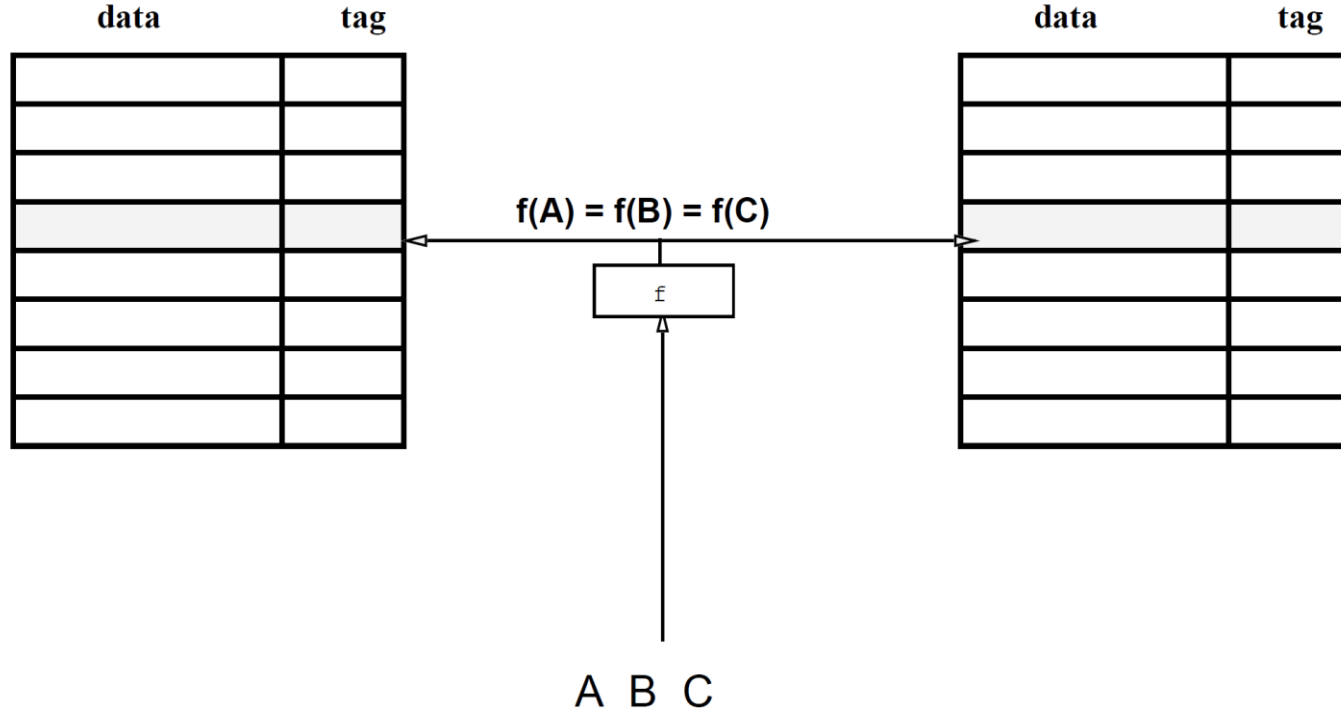
- How to timetable all of DoC and
EEE's 3rd-year, 4th-year and MSc
courses
- With limited number of rooms and
times in the week
- There must be some clashes
- Suppose you want to take two
courses, "ACA" and "BigData"
- If you're lucky they are scheduled
on different slots
- If not, they clash every week!
- Let's rehash every week....

Week 1	Mon@2	ACA	BigData
	Tue@2		
	Wed@2		
	Thu@2		
Week 2	Mon@2	ACA	BigData
	Tue@2		
	Wed@2		
	Thu@2		
Week 3	Mon@2	ACA	BigData
	Tue@2		
	Wed@2		
	Thu@2		
Week 4	Mon@2	ACA	BigData
	Tue@2		
	Wed@2		
	Thu@2		

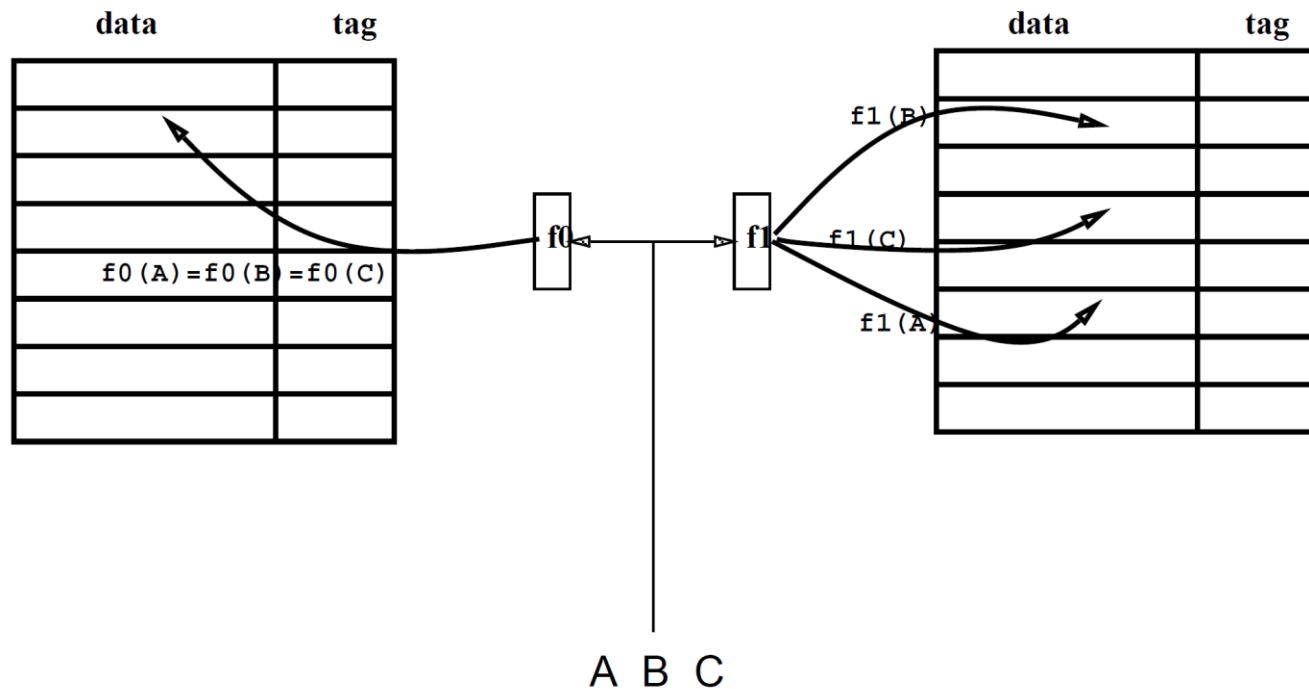
Skewed-associative caches



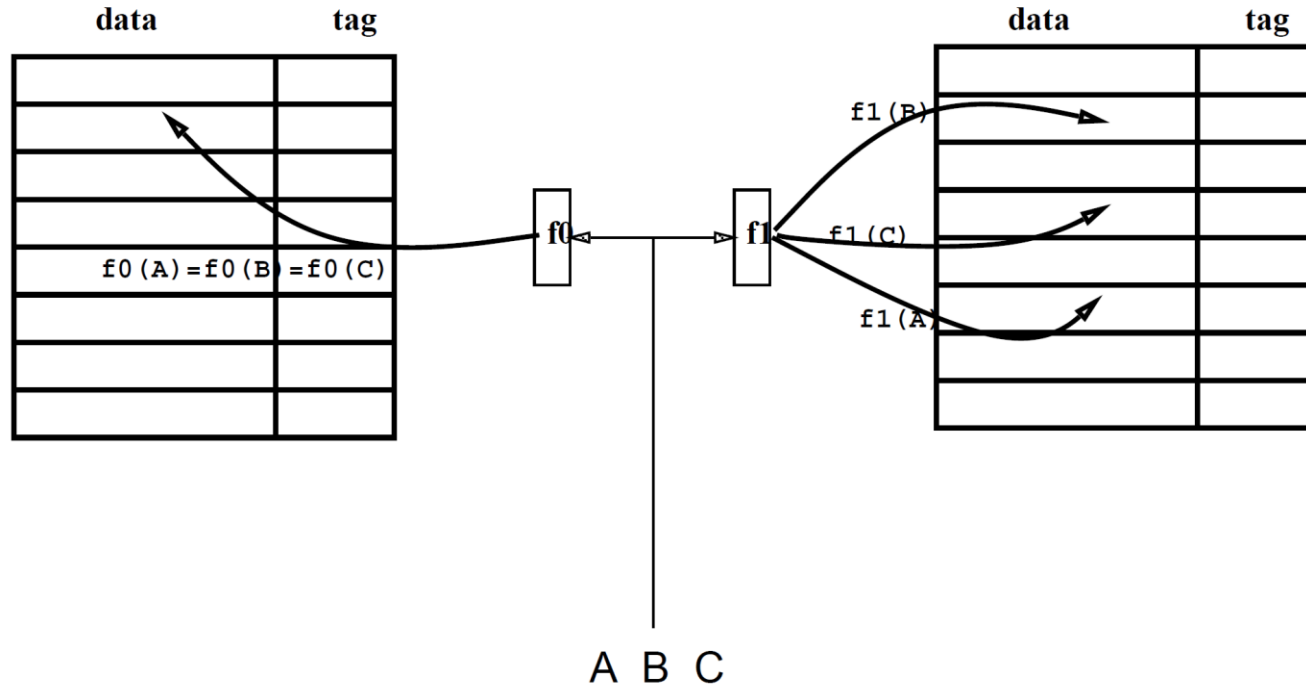
- In a conventional n-way set-associative cache, we get conflicts when $N+1$ blocks have the same address index bits
- Idea: reduce conflict misses by using *different* indices in each cache way
 - We introduce simple hash function,
 - Eg XOR some index bits with tag bits and reorder index bits



**Conventional
two-way set-
associative**



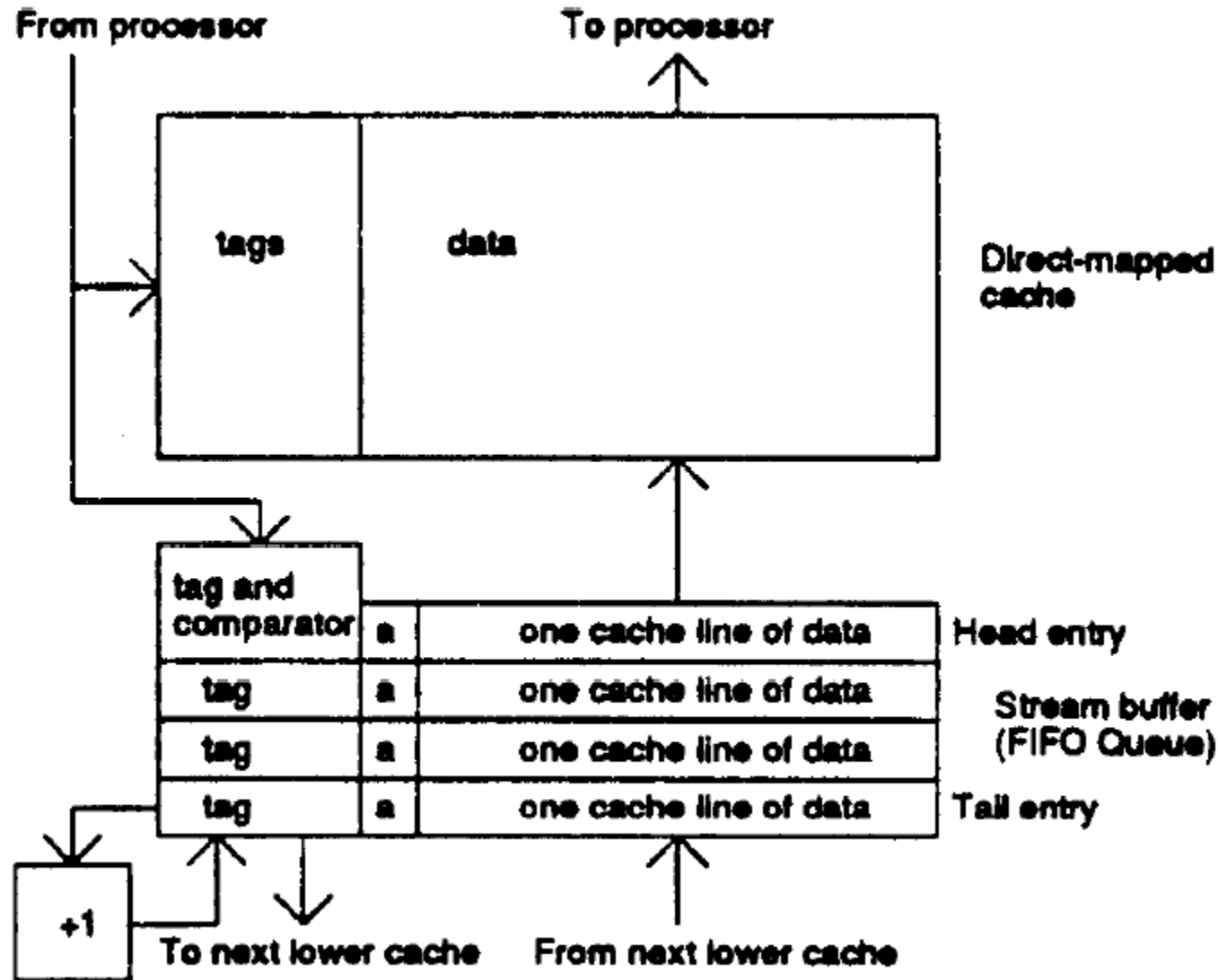
**Skewed
two-way set-
associative**



- If $A[i]$ and $B[i]$ do conflict, it's unlikely that $A[i+1]$ and $B[i+1]$ do
- We may be able to reduce associativity
- We have more predictable *average* performance
- Costs? Latency of hash function, difficulty of implementing LRU, index hash uses *translated* bits

Reducing Misses by Hardware Prefetching of Instructions & Data

- Extra block placed in “stream buffer”
- After a cache miss, stream buffer initiates fetch for *next* block
- But it is not allocated into cache – to avoid “pollution”
- On miss, check stream buffer in parallel with cache
- relies on having extra memory bandwidth

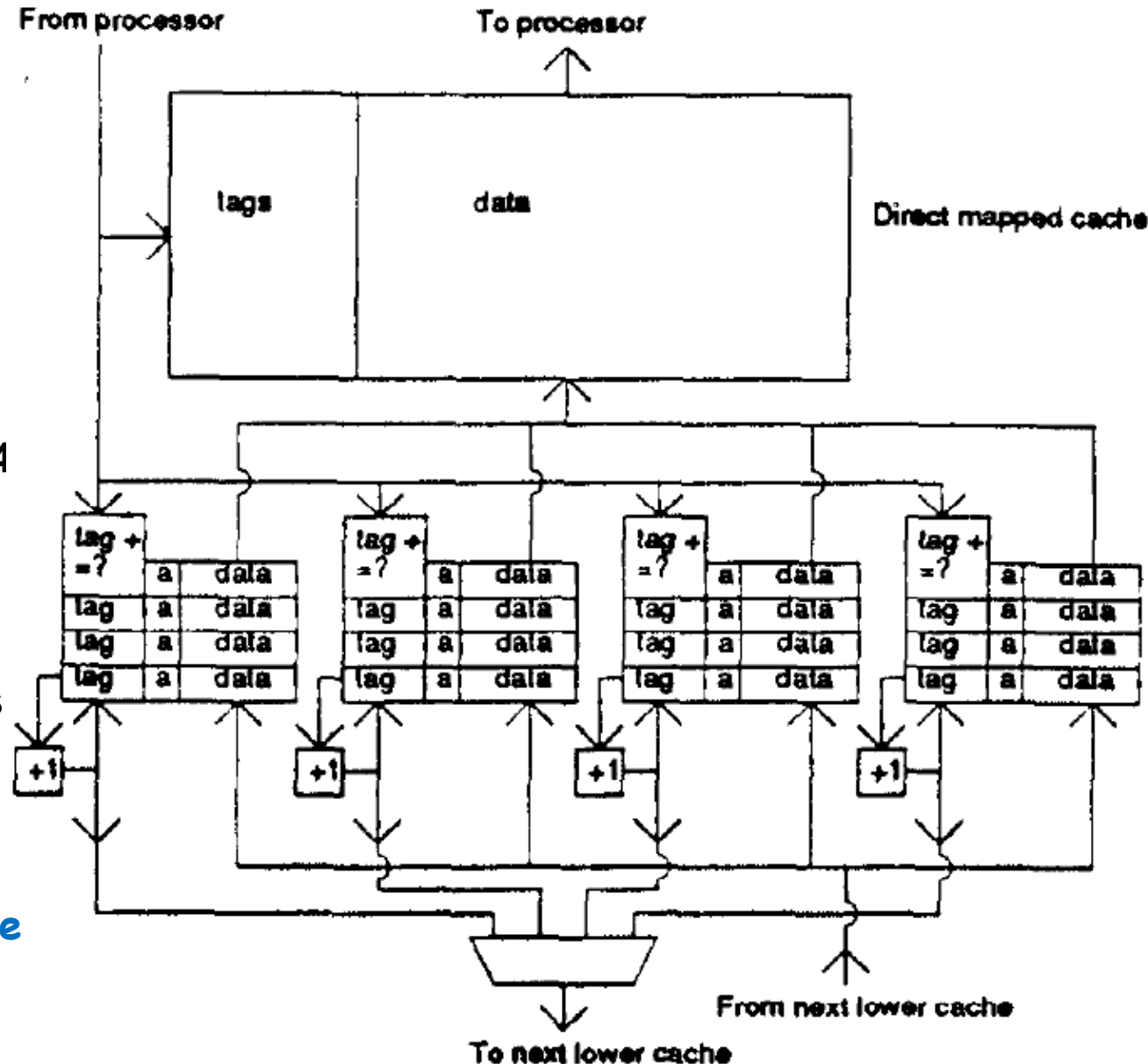


Multi-way stream-buffer

- We can extend this idea to track multiple access streams simultaneously:

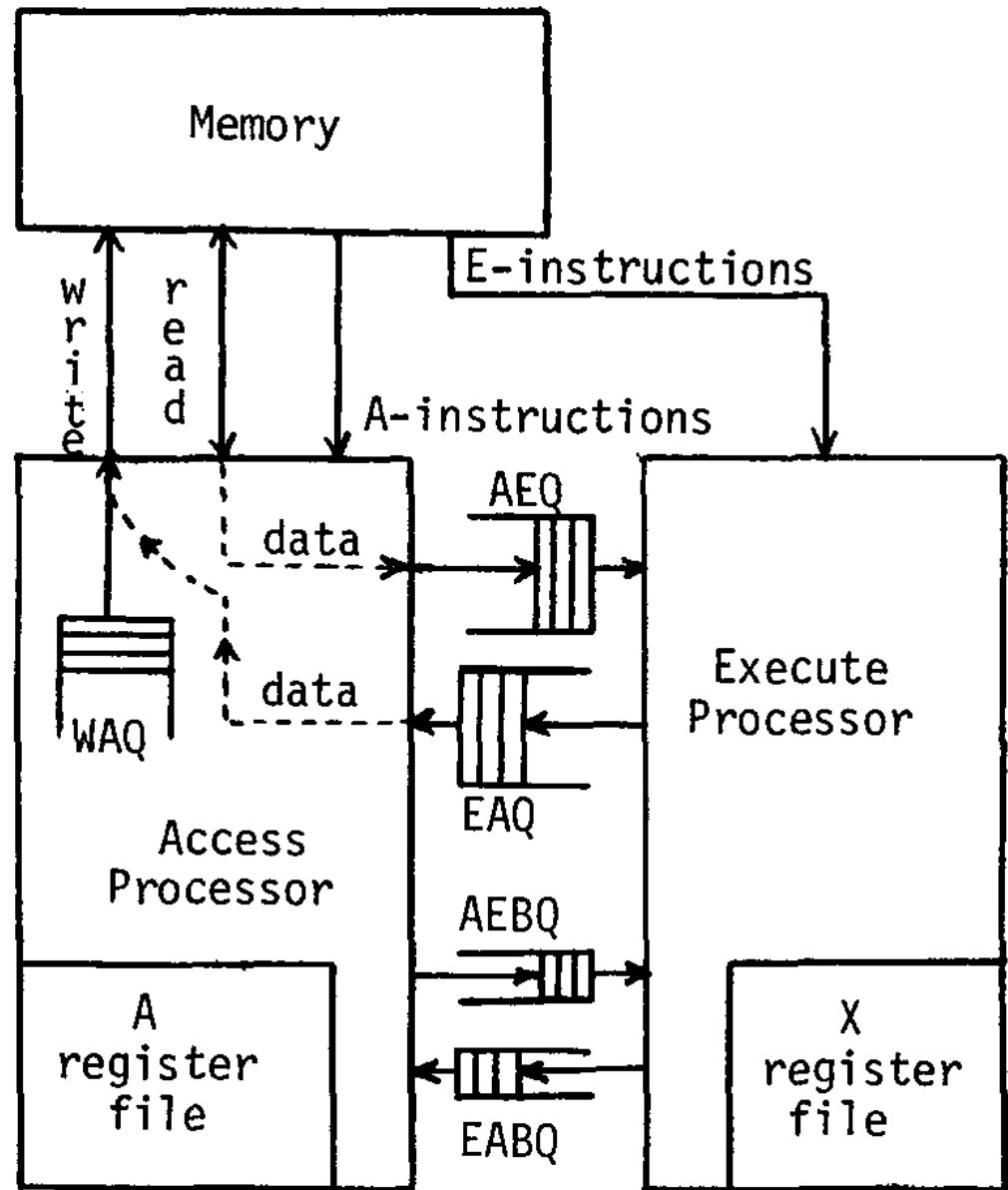
- Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
- Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches

Many (many) modern CPUs have hardware prefetching



Beyond prefetch: decoupled access-execute

- Idea: separate the instructions that generate addresses from the instructions that use memory results
- Let the address-generation side of the machine run ahead



From James E. Smith. 1982. Decoupled access/execute computer architectures. In Proceedings of the 9th annual symposium on Computer Architecture (ISCA '82). IEEE Computer Society Press, Los Alamitos, CA, USA, 112-119

See also ACRI supercomputer project,
<http://www.paralogos.com/DeadSuper/ACRI/>

Reducing Misses by Software Prefetching

● Data Prefetch

- Load data into register (HP PA-RISC loads, Itanium (see later))
- Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
- Special prefetching instructions cannot cause faults; a form of speculative execution

● Prefetching comes in two flavors:

- Binding prefetch: Requests load directly into register.
 - Must be correct address and register! Requires a free register
- Non-Binding prefetch: Load into cache.
 - Can be incorrect. Frees HW/SW to guess! Uses cache capacity

● Issuing Prefetch Instructions takes time

- Is cost of issuing the prefetch instrns < savings in reduced misses?
- Higher superscalar reduces difficulty of issue bandwidth
- But often, hardware prefetching is just as effective

Reducing misses by compiler optimizations

- McFarling [1989]* reduced instruction cache misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
 - By choosing instruction memory layout based on callgraph, branch structure and profile data
 - Reorder procedures in memory so as to reduce conflict misses
 - (actually this really needs the *whole* program – a link-time optimisation)
- Similar (but different) ideas work for data
 - **Merging Arrays**: improve spatial locality by single array of compound elements vs. 2 arrays
 - **Permuting a multidimensional array**: improve spatial locality by matching array layout to traversal order
 - **Loop Interchange**: change nesting of loops to access data in order stored in memory
 - **Loop Fusion**: Combine 2 independent loops that have same looping and some variables overlap
 - **Blocking**: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

* “Program optimization for instruction caches”, ASPLOS89, <http://doi.acm.org/10.1145/70082.68200>

● Storage layout transformations

- **Merging Arrays**: improve spatial locality by single array of compound elements vs. 2 arrays
- **Permuting a multidimensional array**: improve spatial locality by matching array layout to traversal order
- Improve *spatial* locality

● Iteration space transformations

- **Loop Interchange**: change nesting of loops to access data in order stored in memory
- **Loop Fusion**: Combine 2 independent loops that have same looping and some variables overlap
- **Blocking**: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows (wait for Chapter 4)
- Can also improve *temporal locality*

Array Merging - example

```
/* Before: 2 sequential arrays */  
int val[SIZE];  
int key[SIZE];
```

```
/* After: 1 array of structures */  
struct merge {  
    int val;  
    int key;  
};  
struct merge merged_array[SIZE];
```

“Array of Structs” vs
“Struct of Arrays”

(AoS vs SoA)

Reducing conflicts between val & key (example?)

Improve spatial locality (counter-example?)

- **whether this is a good idea depends on access pattern**

(actually this is a transpose: $2 \times \text{SIZE} \rightarrow \text{SIZE} \times 2$)


Consider matrix-matrix multiply (tutorial ex)

- MM1:

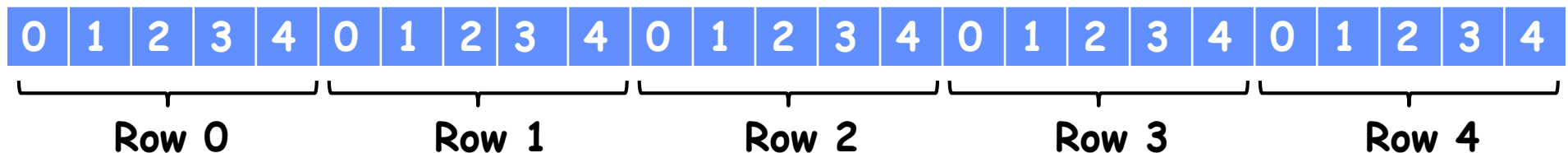
```
for (i=0;i<N;i++)  
  for (j=0;j<N;j++)  
    for (k=0;k<N;k++)  
      C[i][j] += A[i][k] * B[k][j];
```

- MM2:

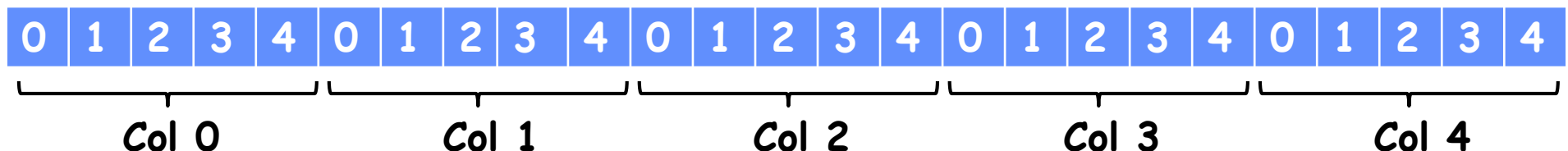
```
for (i=0;i<N;i++)  
  for (k=0;k<N;k++)  
    for (j=0;j<N;j++)  
      C[i][j] += A[i][k] * B[k][j];
```

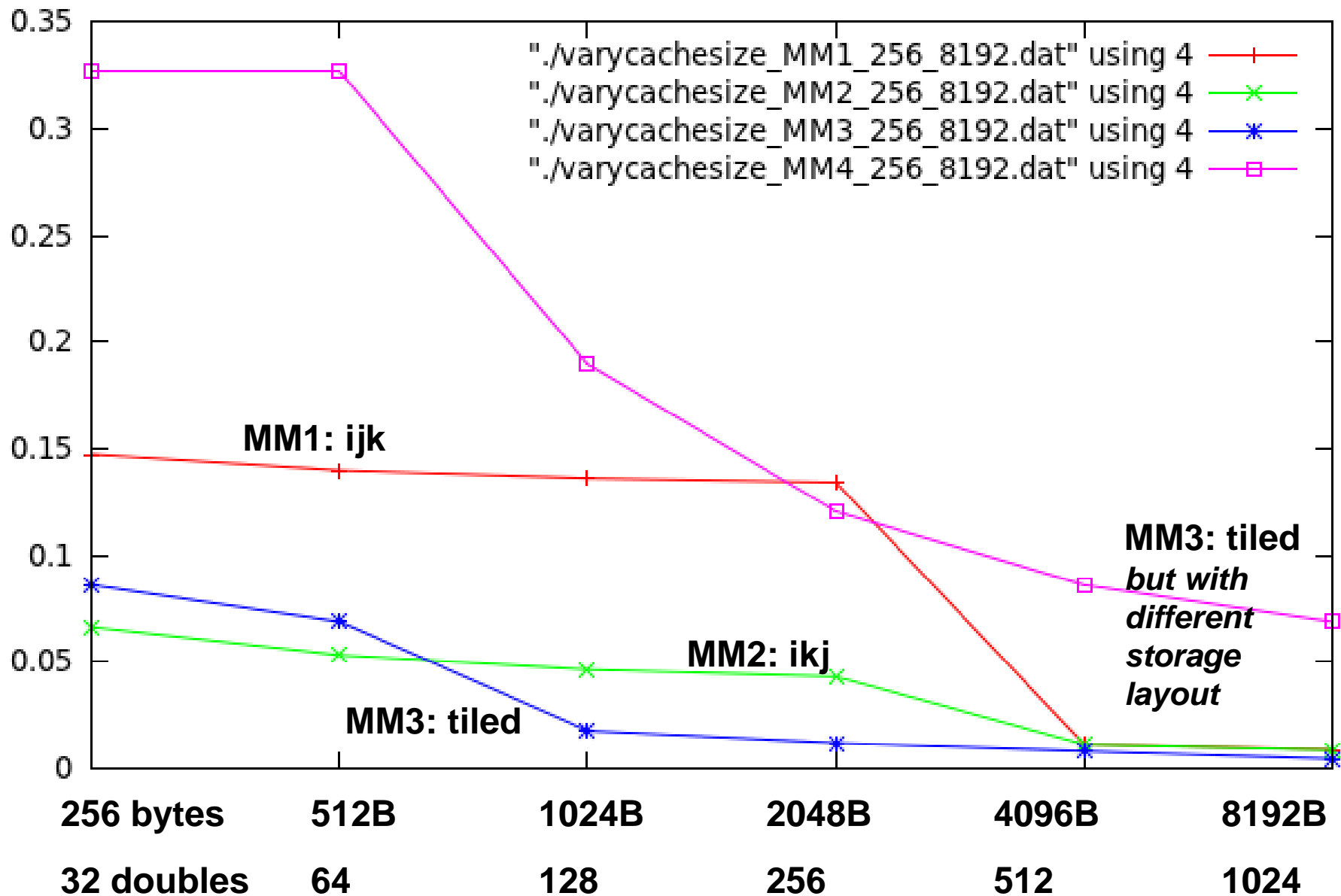


- Row-major storage layout (default for C):



- Column-major storage layout (default for Fortran):

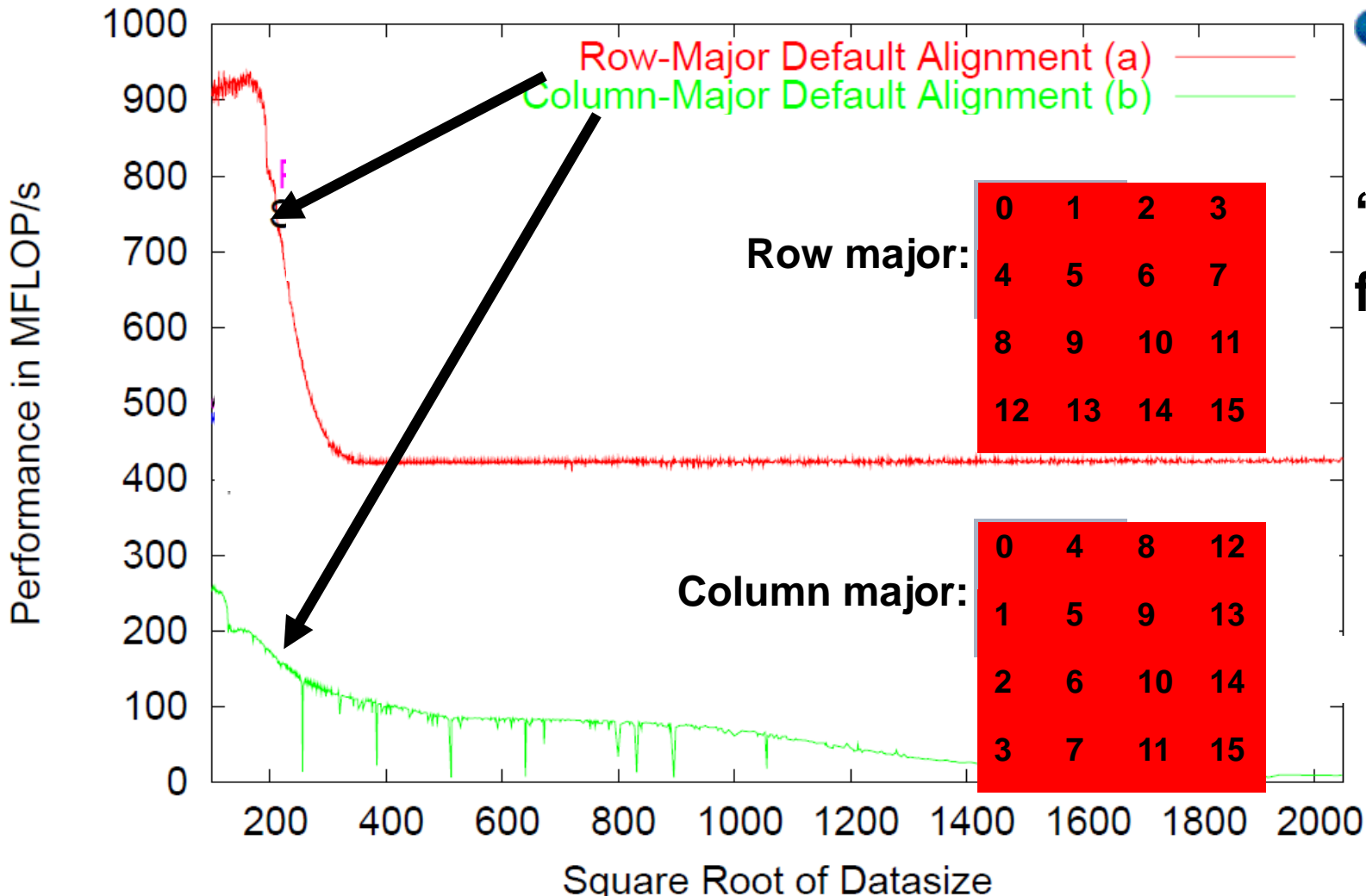




Problem size: 192 doubles, 1536 bytes per row

Permuting multidimensional arrays to improve spatial locality

MMikj on P4: Performance in MFLOP/s



● Matrix-matrix multiply on Pentium 4

“ikj” variant:

for i

for k

for j

$C[ij] += A[ik]$

$* B[kj]$

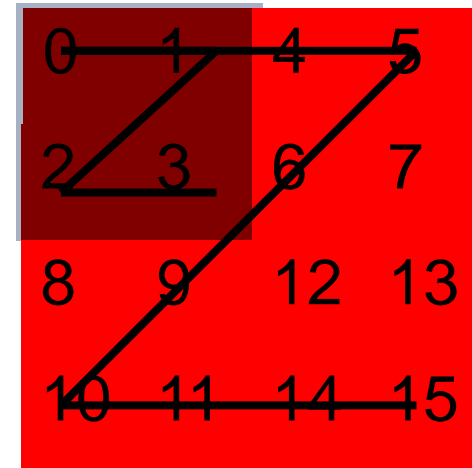
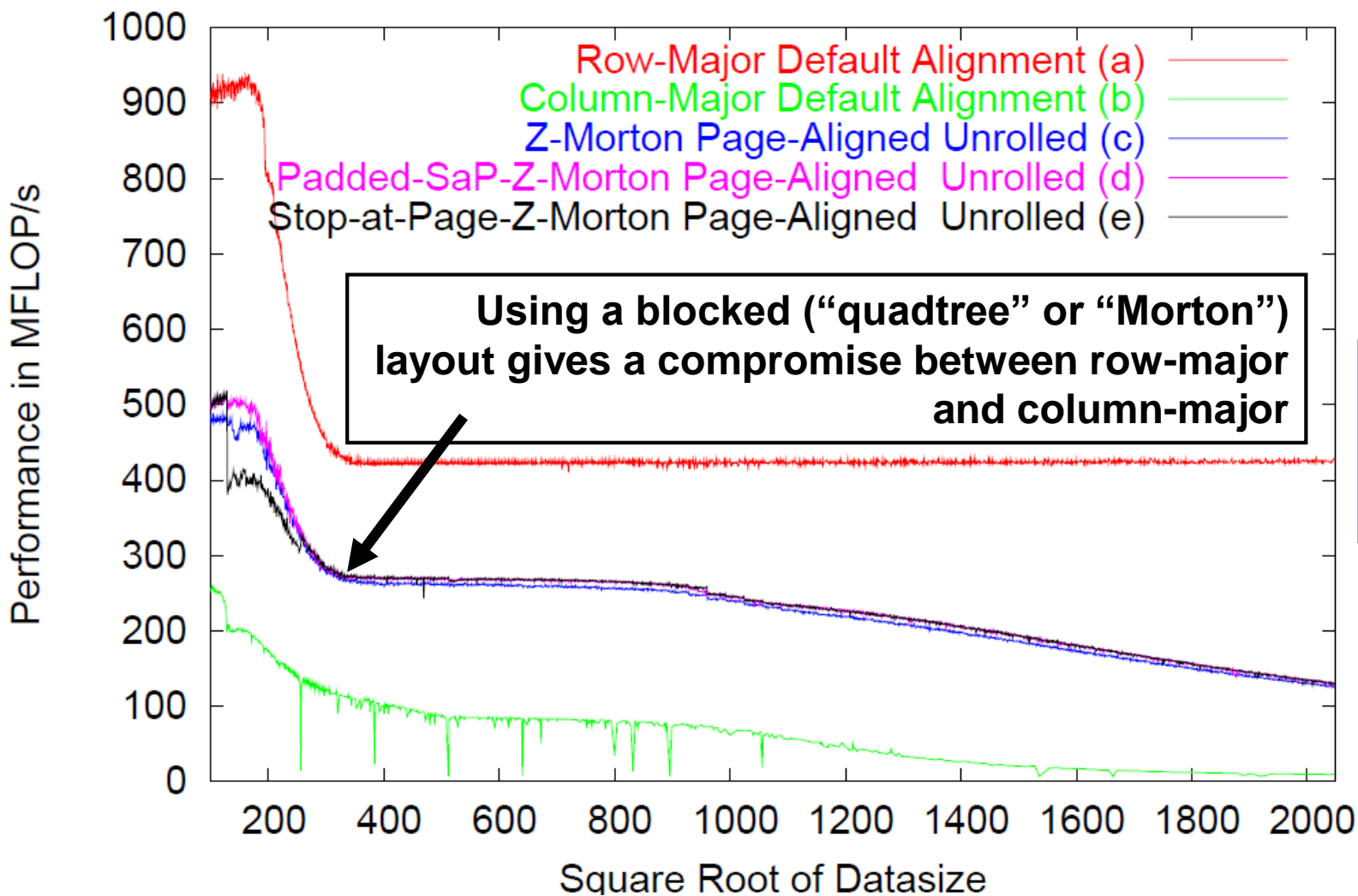
● Traverses B and C in row-major order

● Which is great if the data is stored in row-major order

● If data is actually in column-major order...

Permuting multidimensional arrays to improve spatial locality

MMikj on P4: Performance in MFLOP/s



- Blocked layout offers compromise between row-major and column-major
- Some care is needed in optimising address calculation to make this work (Jeyan Thiyaalingam's Imperial PhD thesis)


Loop Interchange: example

/* Before */

```
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
      x[i][j] = 2 * x[i][j];
```

/* After */

```
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
      x[i][j] = 2 * x[i][j];
```



Sequential accesses: instead of striding through memory every 100 words; improved spatial locality

(actually this is a transpose of the *iteration* space)

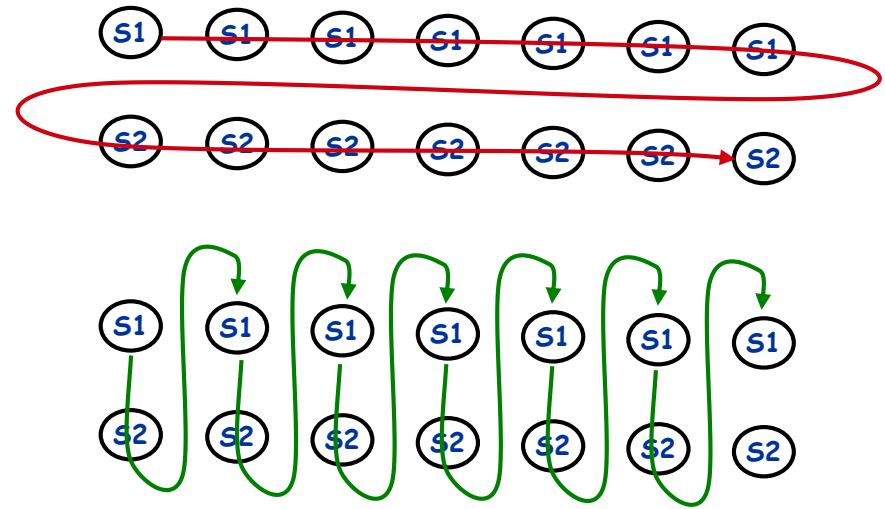
Loop Fusion: example

/* Before */

```
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    S1: a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    S2: d[i][j] = a[i][j] + c[i][j];
```

/* After fusion */

```
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    {S1: a[i][j] = 1/b[i][j] * c[i][j];
     S2: d[i][j] = a[i][j] + c[i][j];}
```



/* After array contraction */

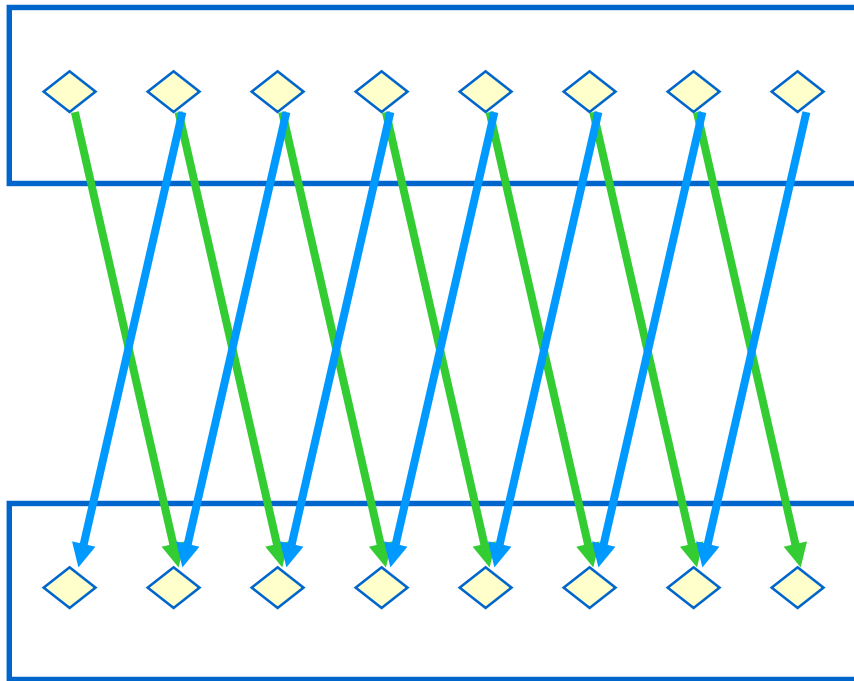
```
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    {  cv = c[i][j];
     S1: a = 1/b[i][j] * cv;
     S2: d[i][j] = a + cv;}
```

2 misses per access to a & c vs.
one miss per access; improve
spatial locality

The real payoff comes if
fusion enables **Array
Contraction**: values
transferred in scalar
instead of via array

Fusion is not always so simple

- Dependences might not align nicely
- Example: one-dimensional filters



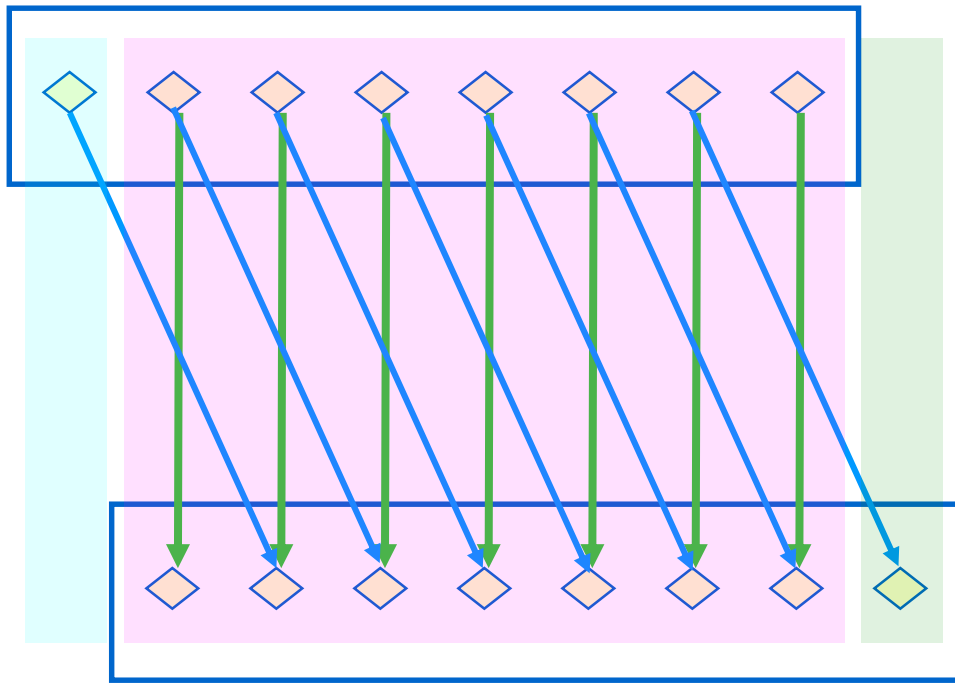
```
for (i=1; i<N; i++)  
    V[i] = (U[i-1] + U[i+1])/2
```

```
for (i=1; i<N; i++)  
    W[i] = (V[i-1] + V[i+1])/2
```

- “Stencil” loops are not directly fusable

Loop fusion – code expansion

- We make them fusable by shifting:



$$V[1] = (U[0] + U[2])/2$$

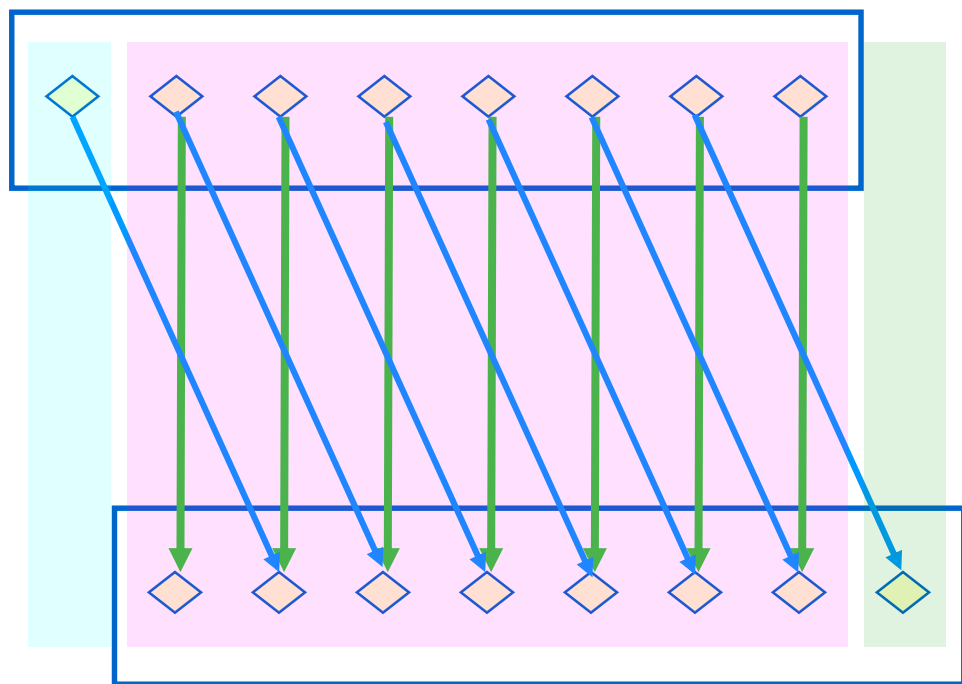
```
for (i=2; i<N; i++) {  
    V[i] = (U[i-1] + U[i+1])/2  
    W[i-1] = (V[i-2] + V[i])/2  
}
```

$$W[N-1] = (V[N-2] + V[N])/2$$

- The middle loop is fusable
- We get lots of little edge bits

Loop fusion – code expansion

- We make them fusable by shifting:



- The middle loop is fusable
- We get lots of little edge bits

$$V[1] = (U[0] + U[2])/2$$

```
for (i=2; i<N; i++) {  
    V[i%4] = (U[i-1] + U[i+1])/2  
    W[i-1] = (V[(i-2)%4] + V[i%4])/2  
}
```

$$W[N-1] = (V[(N-2)\%4] + V[N\%4])/2$$

- Contraction is trickier
- We need the last *two* Vs
- We need 3 V locations
- Quicker to round up to four

Summary: Miss Rate Reduction

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times \text{Miss rate} \times Miss\ penalty \right) \times Clock\ cycle\ time$$

● 3 Cs: Compulsory, Capacity, Conflict

1. Reduce Misses via Larger Block Size
2. Reduce Misses via Higher Associativity
3. Reducing Misses via Victim Cache
4. Reducing Misses via Pseudo-Associativity
5. “Reducing” Misses via skewed associativity
6. Reducing Misses by HW Prefetching - Instructions, or Data
7. Reducing Misses by SW Prefetching Data (or instructions?)
8. Reducing Misses by Compiler Optimizations

● Prefetching comes in two flavors:

- Binding prefetch: Requests load directly into register.
 - Must be correct address and register!
- Non-Binding prefetch: Load into cache.
 - Can be incorrect. Frees HW/SW to guess!

Average memory access time:

$$\text{AMAT} = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

There are three ways to improve cache performance:

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache

We now look at each of these in turn...

Write Policy: Write-Through vs Write-Back

- **Write-through: all writes update cache and underlying memory/cache**

- Can always discard cached data - most up-to-date data is in memory
- Cache control bit: only a *valid* bit

- **Write-back: all writes simply update cache**

- Can't just discard cached data - may have to write it back to memory
- Cache control bits: both *valid* and *dirty* bits

- **Other Advantages:**

- **Write-through:**

- memory (or other processors) always have latest data
- Simpler management of cache

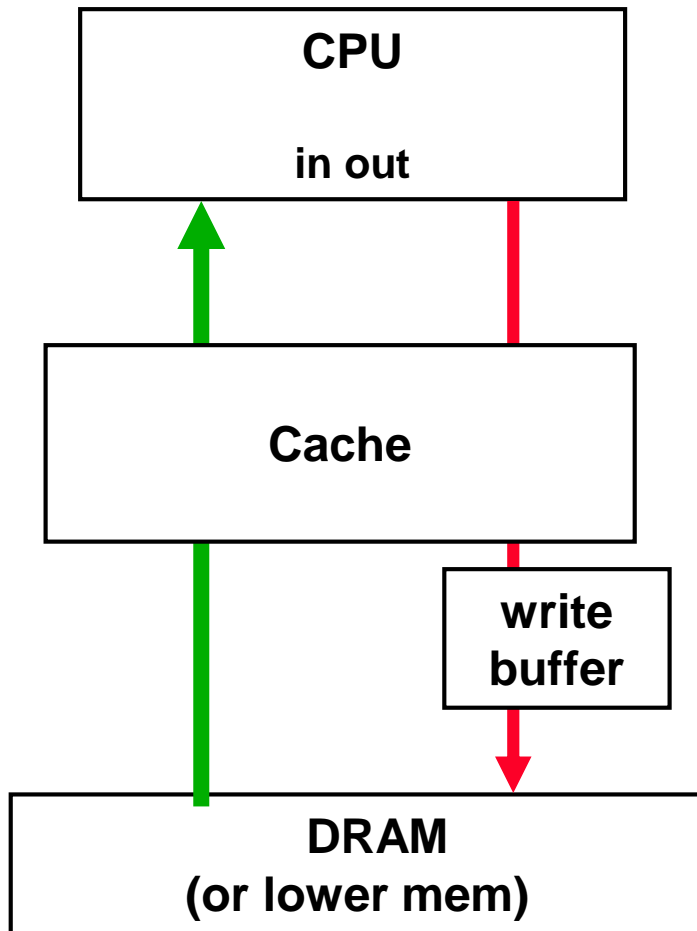
- **Write-back:**

- much lower bandwidth, since data often overwritten multiple times
- Better tolerance to long-latency memory?

Write Policy 2: Write Allocate vs Non-Allocate (What happens on write-miss)

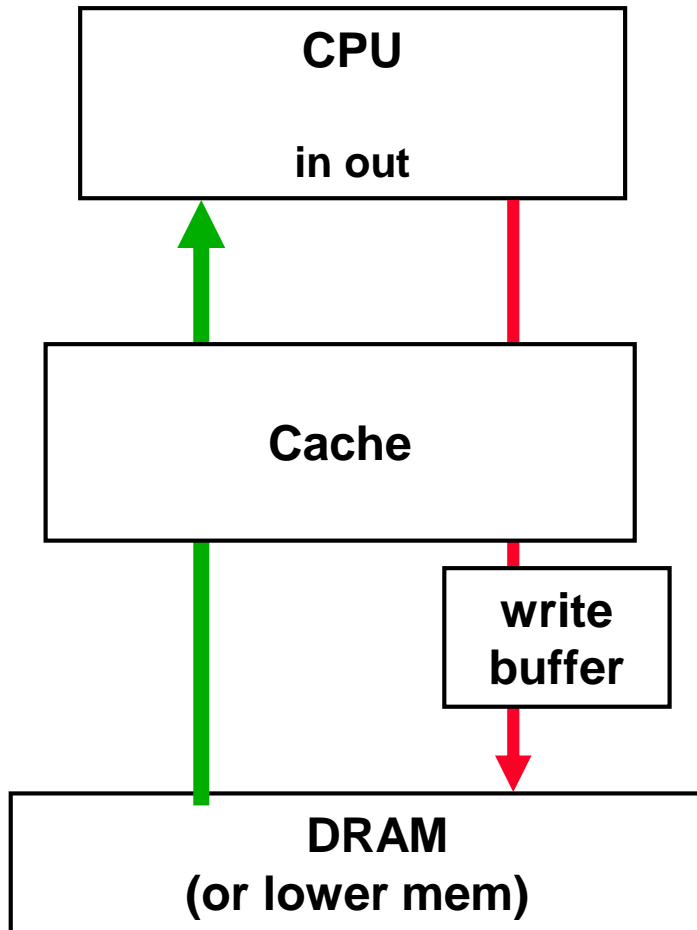
- **Write allocate: allocate new cache line in cache**
 - Usually means that you have to do a “read miss” to fill in rest of the cache-line!
 - Alternative: per/word valid bits
- **Write non-allocate (or “write-around”):**
 - Simply send write data through to underlying memory/cache - don't allocate new cache line!

1. Reducing Miss Penalty: Read Priority over Write on Miss



- Consider write-through with write buffers
 - RAW conflicts with main memory reads on cache misses
 - Could simply wait for write buffer to empty, before allowing read
 - Risks serious increase in read miss penalty (old MIPS 1000 by 50%)
 - Solution:
 - Check write buffer contents before read; if no conflicts, let the memory access continue
- Write-back also needs buffer to hold displaced blocks
 - Read miss replacing dirty block
 - Normal: Write dirty block to memory, and then do the read
 - Instead copy the dirty block to a write buffer, then do the read, and then do the write
 - CPU stall less since restarts as soon as do read

Write buffer issues



- Size: 2-8 entries are typically sufficient for caches
 - But an entry may store a whole cache line
 - Make sure the write buffer can handle the typical store bursts...
 - Analyze your common programs, consider bandwidth to lower level
- Coalescing write buffers
 - Merge adjacent writes into single entry
 - Especially useful for write-through caches
- Dependency checks
 - Comparators that check load address against pending stores
 - If match there is a dependency so load must stall
- Optimization: load forwarding
 - If match and store has its data, forward data to load...
- Integrate with victim cache?

2. Reduce Miss Penalty: Early Restart and Critical Word First

- Don't wait for full block to be loaded before restarting CPU
 - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
 - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called *wrapped fetch* and *requested word first*
- Generally useful only in large blocks,
- (Access to contiguous sequential words is very common – but doesn't benefit from either scheme – are they worthwhile?)



block

3. Reduce Miss Penalty: Non-blocking Caches to reduce stalls on misses

- Non-blocking cache or lockup-free cache allows data cache to continue to supply cache hits during a miss
 - requires full/empty bits on registers or out-of-order execution
 - requires multi-bank memories
- “hit under miss” reduces the effective miss penalty by working during miss instead of ignoring CPU requests
- “hit under multiple miss” or “miss under miss” may further lower the effective miss penalty by overlapping multiple misses
 - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
 - Requires multiple memory banks (otherwise cannot support)
 - Eg IBM Power5 allows 8 outstanding cache line misses

Compare:

prefetching: overlap memory access with pre-miss instructions,

Non-blocking cache: overlap memory access with post-miss instructions

What happens on a Cache miss?

- For in-order pipeline, two options:

- Freeze pipeline in Mem stage (popular early on: Sparc, R4000)

IF	ID	EX	Mem	stall	stall	stall	...	stall	Mem	Wr
	IF	ID	EX	stall	stall	stall	...	stall	stall	Ex Wr

- Use Full/Empty bits in registers + MSHR queue

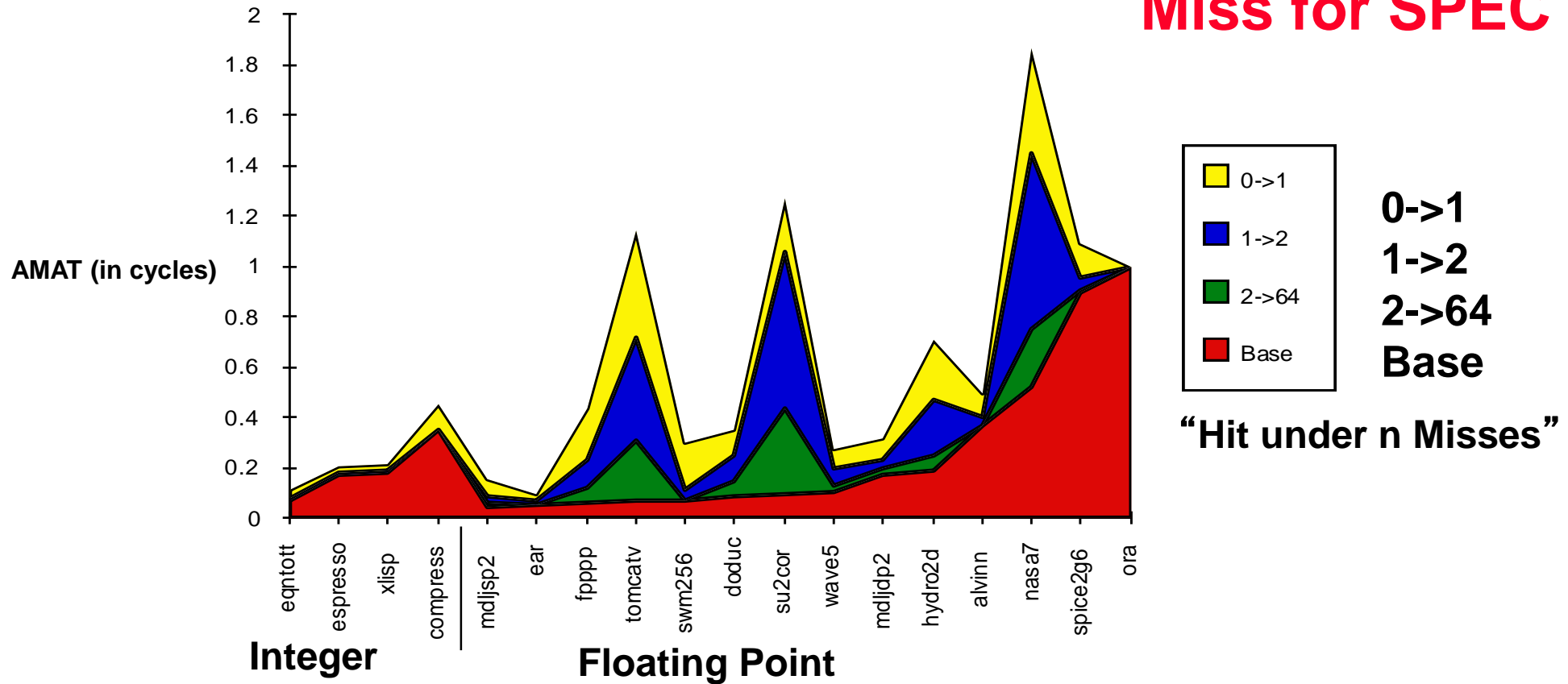
- MSHR = “Miss Status/Handler Registers” (Kroft*)

Each entry in this queue keeps track of status of outstanding memory requests to one complete memory line.

- Per cache-line: keep info about memory address.
- For each word: register (if any) that is waiting for result.
- Used to “merge” multiple requests to one memory line
- New load creates MSHR entry and sets destination register to “Empty”. Load is “released” from pipeline.
- Attempt to use register before result returns causes instruction to block in decode stage.
- Limited “out-of-order” execution with respect to loads.

Popular with in-order superscalar architectures.

- Out-of-order pipelines already have this functionality built in... (load queues, etc). Cf also Power6 “load lookahead mode”



- FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26

- Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19

- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss

- Hit-under-miss implies loads may be serviced out-of-order...**

- Need a memory “fence” or “barrier” (<http://www.linuxjournal.com/article/8212>)

- PowerPC eieio (Enforce In-order Execution of Input/Output) Instruction**

4: Add a second-level cache

● L2 Equations

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})$$

● Definitions:

- **Local miss rate**—misses in this cache divided by the total number of memory accesses **to this cache** (Miss rate_{L2})
- **Global miss rate**—misses in this cache divided by the total number of memory accesses **generated by the CPU** ($\text{Miss Rate}_{L1} \times \text{Miss Rate}_{L2}$)
- Global Miss Rate is what matters

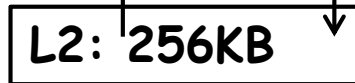
Multiple levels of cache - example



L1: 32KB, 8-way associative I and D

L1D: writeback, two 256-bit loads and a 256-bit store every cycle

64B/cycle 32B/cycle



L2: 256KB, 8-way writeback with ECC. Can provide a full 64B line to the data or instruction cache every cycle, 11 cycle minimum latency and 16 outstanding misses.

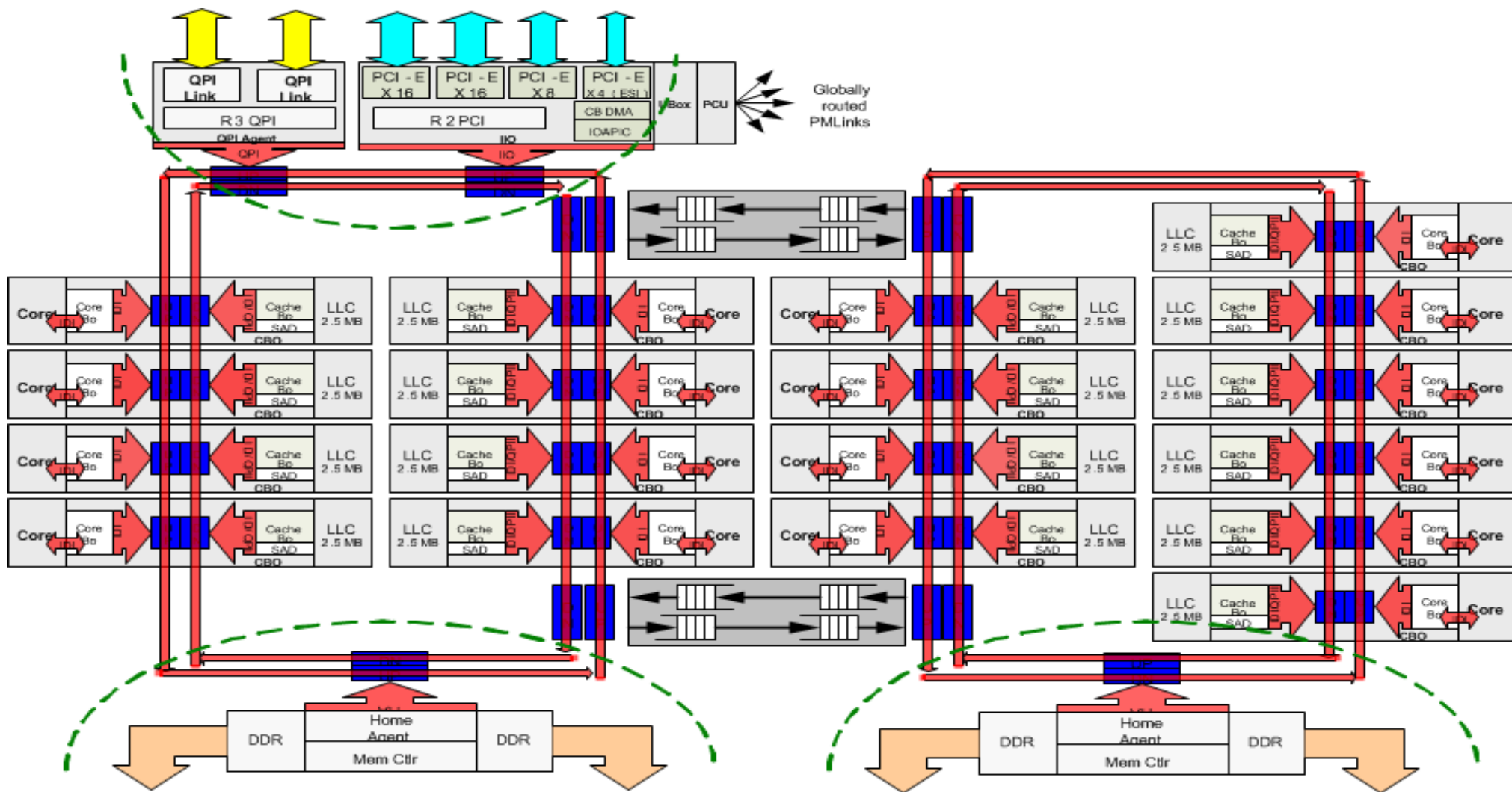
32B/cycle



L3: Size varies from device to device. Shared by all cores on chip. Connected by ring interconnect (actually two connected rings)



- Example: Intel Haswell e5 2600 v3
- 18 cores, 145W TDP, 5.56B transistors



- Example: Intel Haswell e5 2600 v3
- Q: do all LLC hits have same latency?
- Q: do all LLC misses have same latency?

● Multi-level inclusion

- L2 cache contains everything in L1
- L_{n+1} cache contains everything in L_n

Multilevel inclusion

- *We might allocate into L1 but not into L2*
- *We might allocate into L2 but not into L1*
- *We might allocate into L1 and L2 but not LLC*

- L3 (Last-level cache) is sometimes managed as a victim cache – data is allocated into LLC when displaced from L2 (eg AMD Barcelona, Apple A9)
- Example: Intel's Crystalwell processor has a 128MB DRAM L4 cache on a separate chip in the same package as the CPU, managed as a victim cache

● Issues:

- replacement of dirty lines?
- Cache coherency - invalidation
 - With MLI, if line is not in L2, we don't need to invalidate it in L1

Average memory access time:

$$AMAT = HitTime + MissRate \times MissPenalty$$

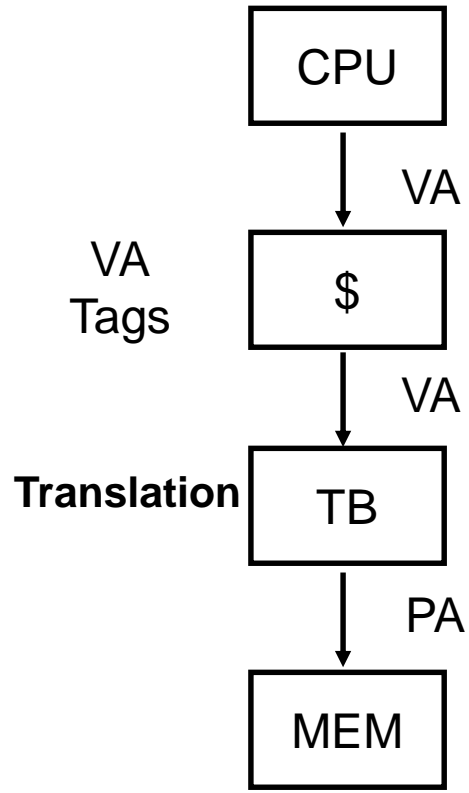
There are three ways to improve cache performance:

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

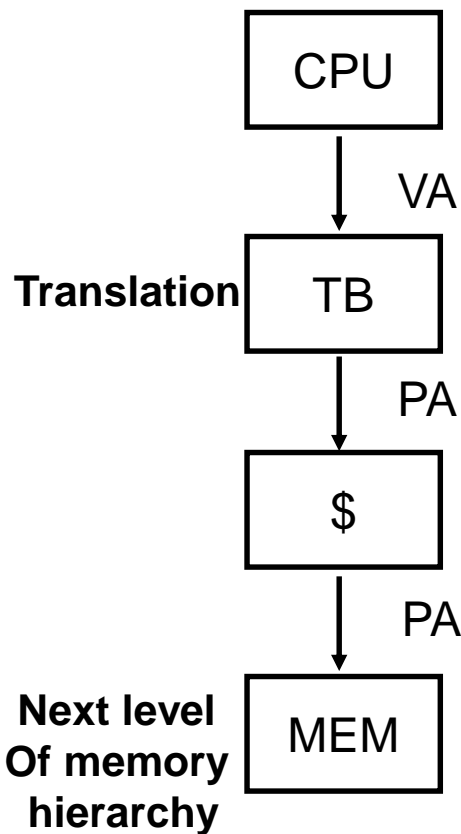
Virtual memory, and address translation

- Simple processors access memory directly
 - Addresses generated by the processor are used directly to access memory
- What if you want to
 - Run some code in an isolated environment
 - So that if it fails it won't crash the whole system
 - So that if it's malicious it won't have total access
 - Run more than one application at a time
 - So they can't interfere with each other
 - So they don't need to know about each other
 - Use more memory than DRAM
- An effective solution to this is to *virtualise* the addressing of memory
- By adding address translation

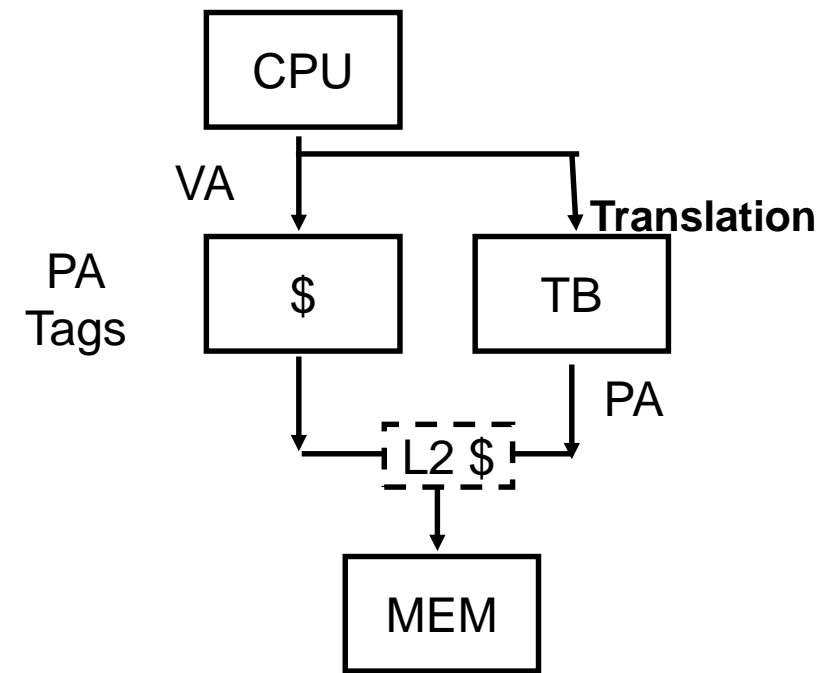
2. Fast hits by Avoiding Address Translation



Virtually-indexed, virtually tagged (**VIVT**): Translate only on miss
Synonym/homonym problems



Physically-indexed,
Physically-tagged (**PIPT**)

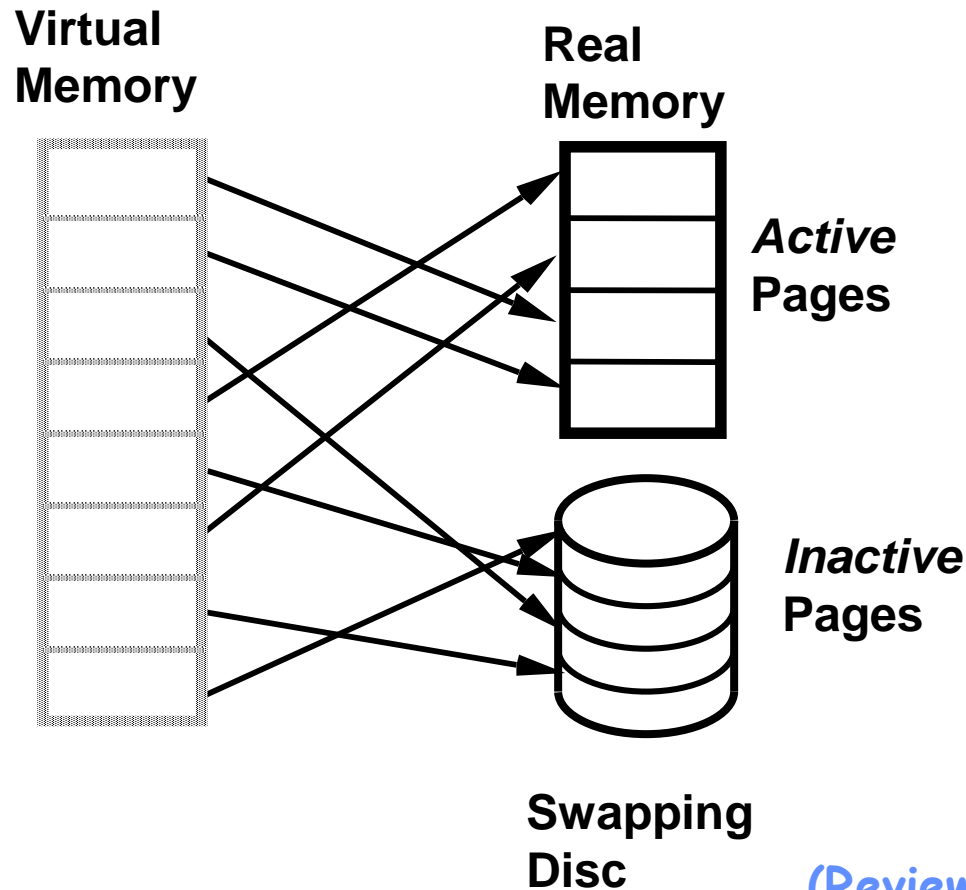


Virtually-indexed, physically-tagged (**VIPT**)
Overlap \$ access with VA translation:
requires \$ index to remain invariant across translation

- **CPU issues Virtual Addresses (VAs)**
- **TB translates Virtual Addresses to Physical Addresses (PAs)**

Paging

Virtual address space is divided into **pages** of equal size.
Main Memory is divided into **page frames** the same size.



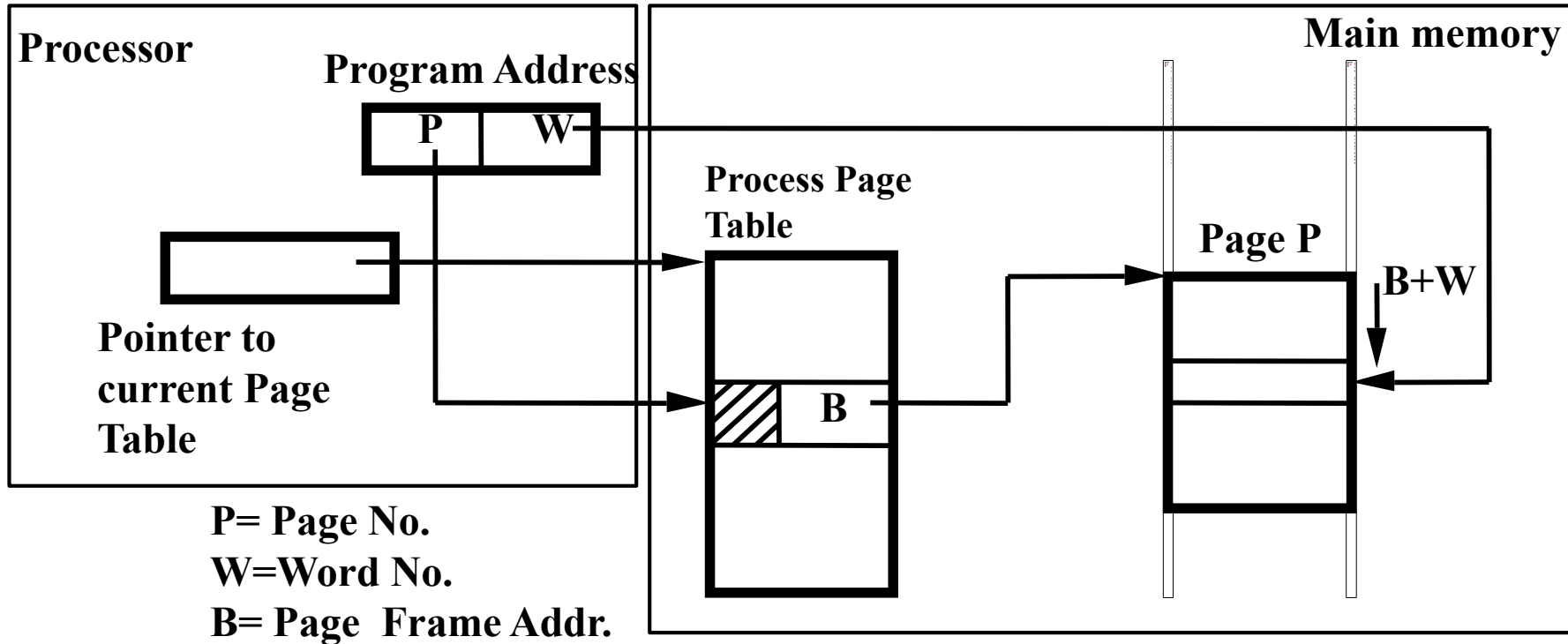
- Running or ready process
 - some pages in main memory
- Waiting process
 - all pages can be on disk
- Paging is transparent to programmer

Paging Mechanism

- (1) Address Mapping
- (2) Page Transfer

(Review introductory operating systems material
for students lacking CS background)

Paging - Address Mapping



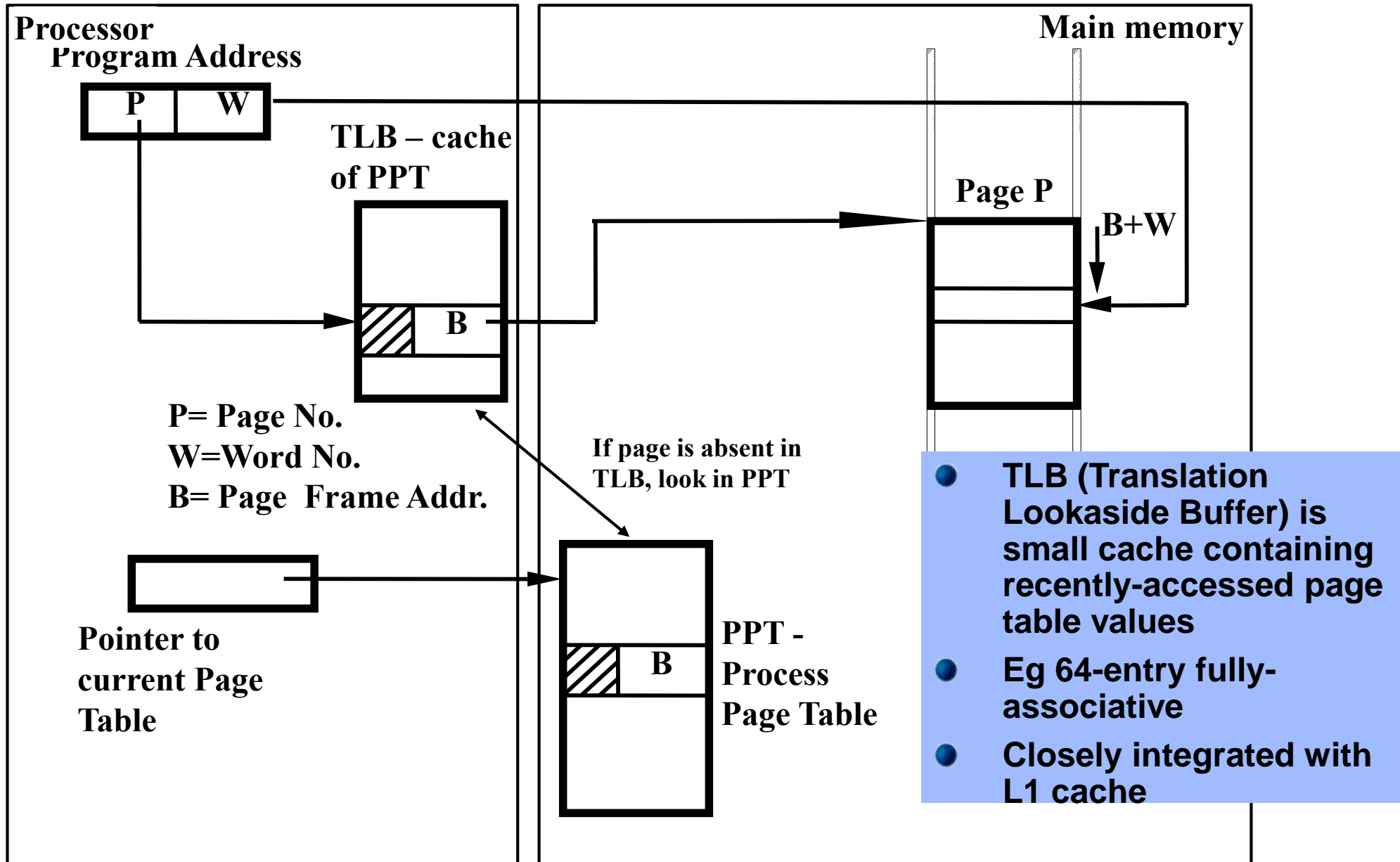
Example: Word addressed machine, $W = 8$ bits, page size = 4096

$$\text{Amap}(P, W) := \text{PPT}[P] * 4096 + W$$

Note: The Process Page Table (PPT) itself can be paged

**(Review introductory operating systems material
for students lacking CS background)**

Paging - Address Mapping



Paging - Page Transfer

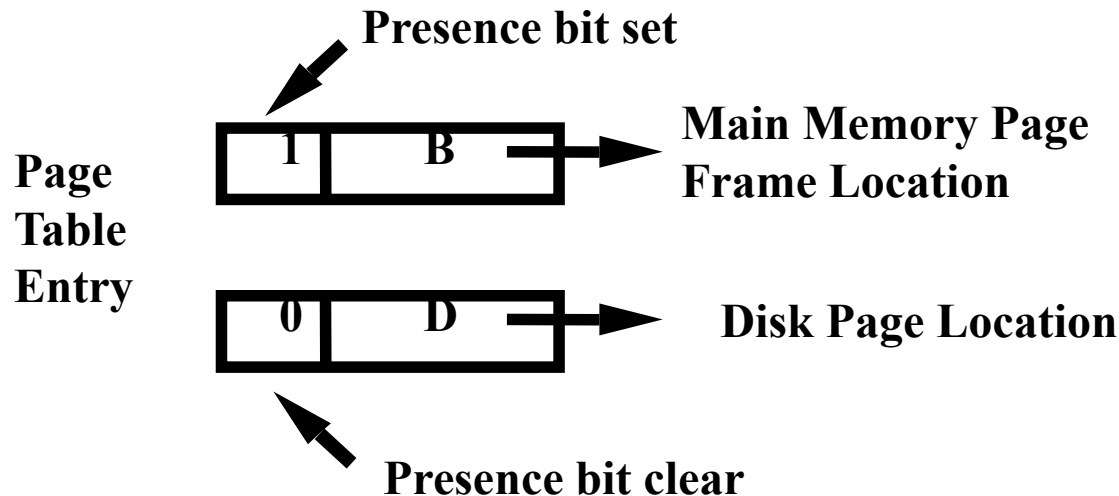
What happens when we access a page which is currently not in main memory (i.e. the page table entry is empty)?

[?] Page Fault

- Suspend running process
- Get page from disk
- Update page table
- Resume process (re-execute instruction)

? Can one instruction cause more than one page fault?

The location of a page on disk can be recorded in a separate table or in the page table itself using a ***presence bit***.



Note: We can run another *ready* process while the page fault is being serviced.

(Review introductory operating systems material for students lacking CS background)

Synonyms and homonyms in address translation

● Homonyms (*same sound different meaning*)

- same virtual address points to two different physical addresses in different processes
- If you have a virtually-indexed cache, flush it between context switches - or include a process identifier (PID) in the cache tag

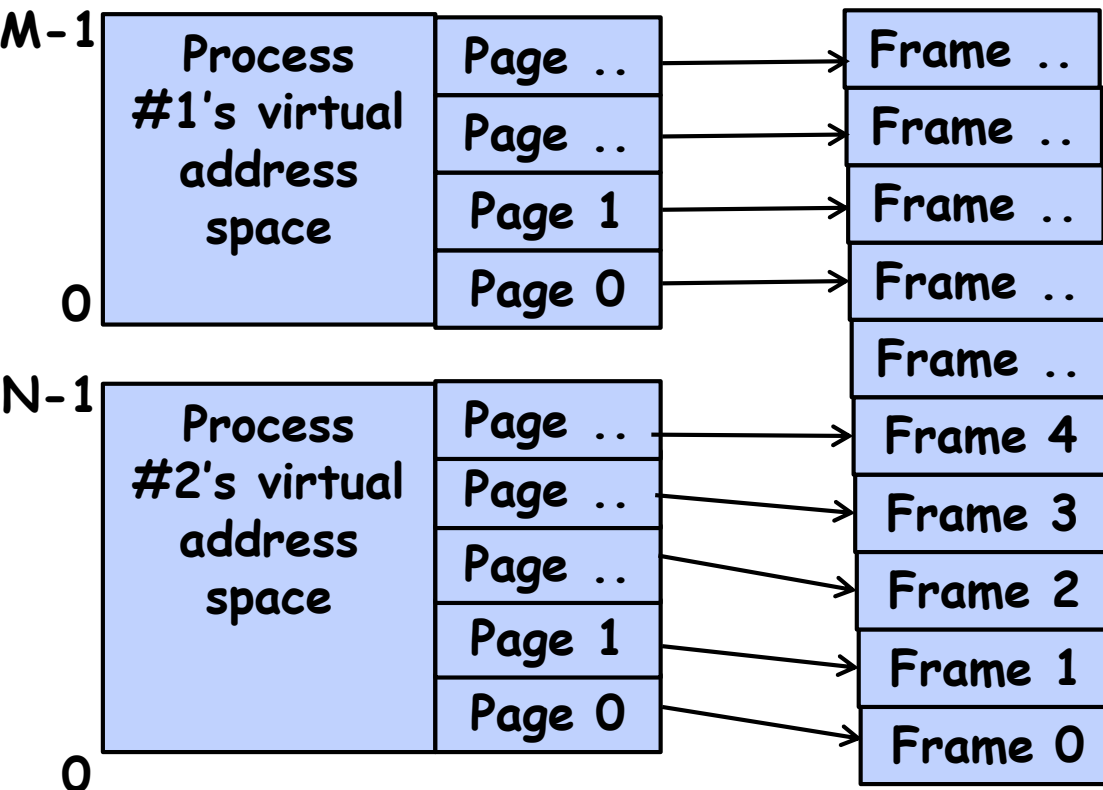
● Synonyms (*different sound same meaning*)

- different virtual addresses (from the same or different processes) point to the same physical address
- in a virtually addressed cache
 - a virtual address could be cached twice under different physical addresses
 - updates to one cached copy would not be reflected in the other cached copy
 - solution: make sure synonyms can't co-exist in the cache, e.g., OS *can force synonyms to have the same index bits in a direct mapped cache* (sometimes called page colouring)

(a nice explanation in more detail can be found at <http://www.ece.cmu.edu/~jhoef/course/ece447/handouts/L22.pdf>)

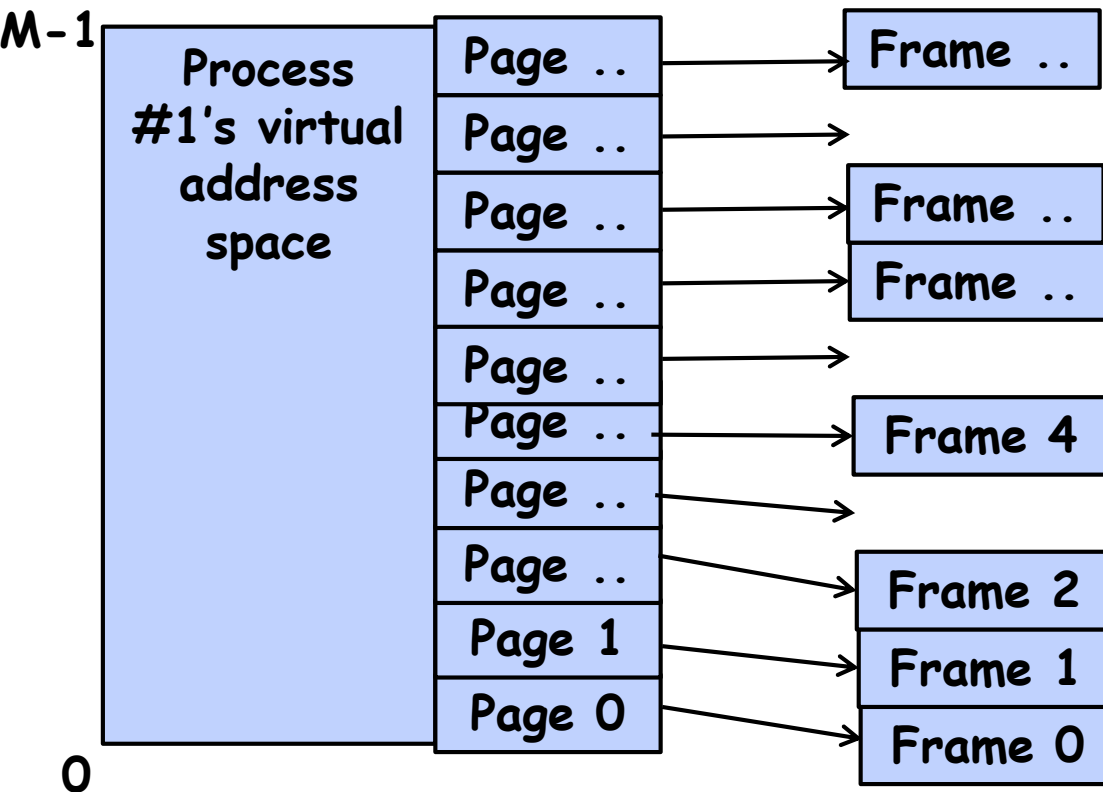
Maybe also see <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344h/BEIBFJEA.html>

What address translation is for



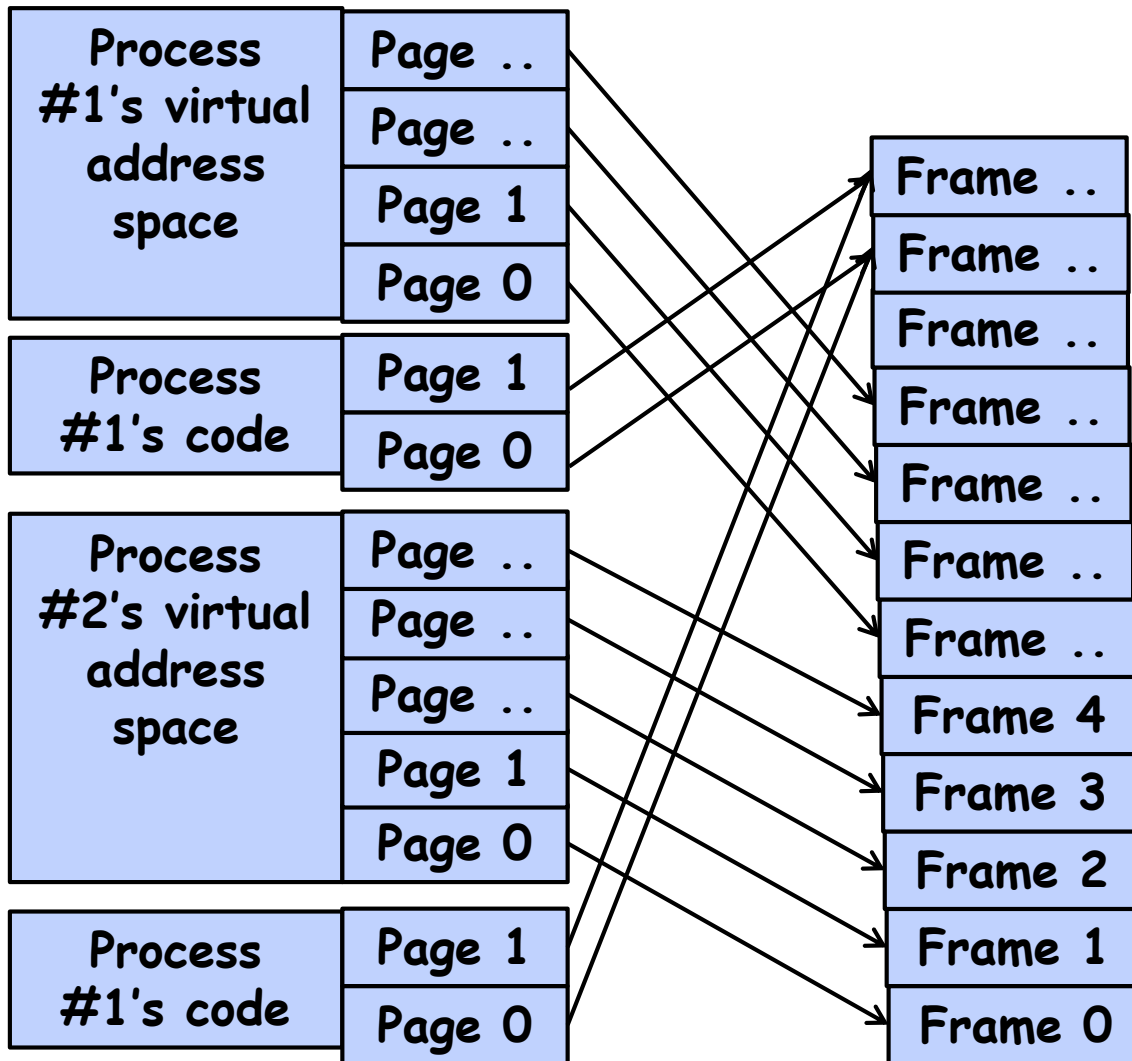
- Two different processes sharing the same physical memory

What address translation is for



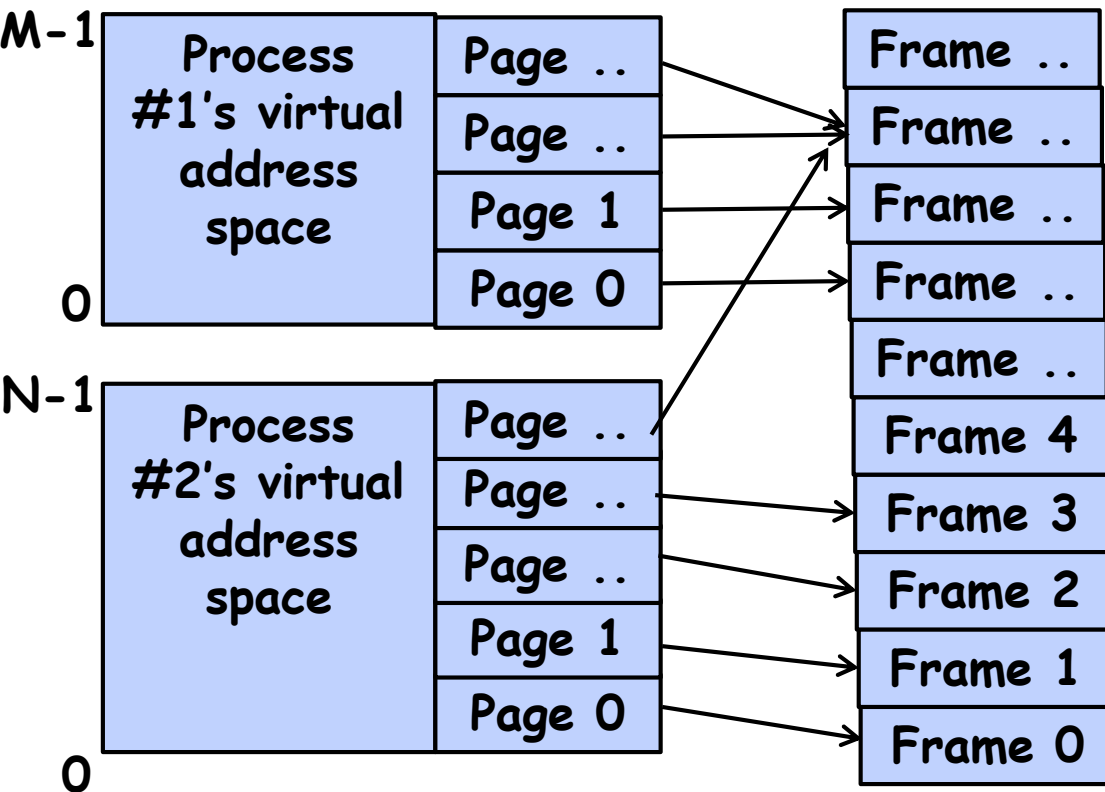
- Virtual address space may be larger than physical address space
- Some pages may be absent – OS (re-)allocates when fault occurs
- Virtual address space may be very large

What address translation is for



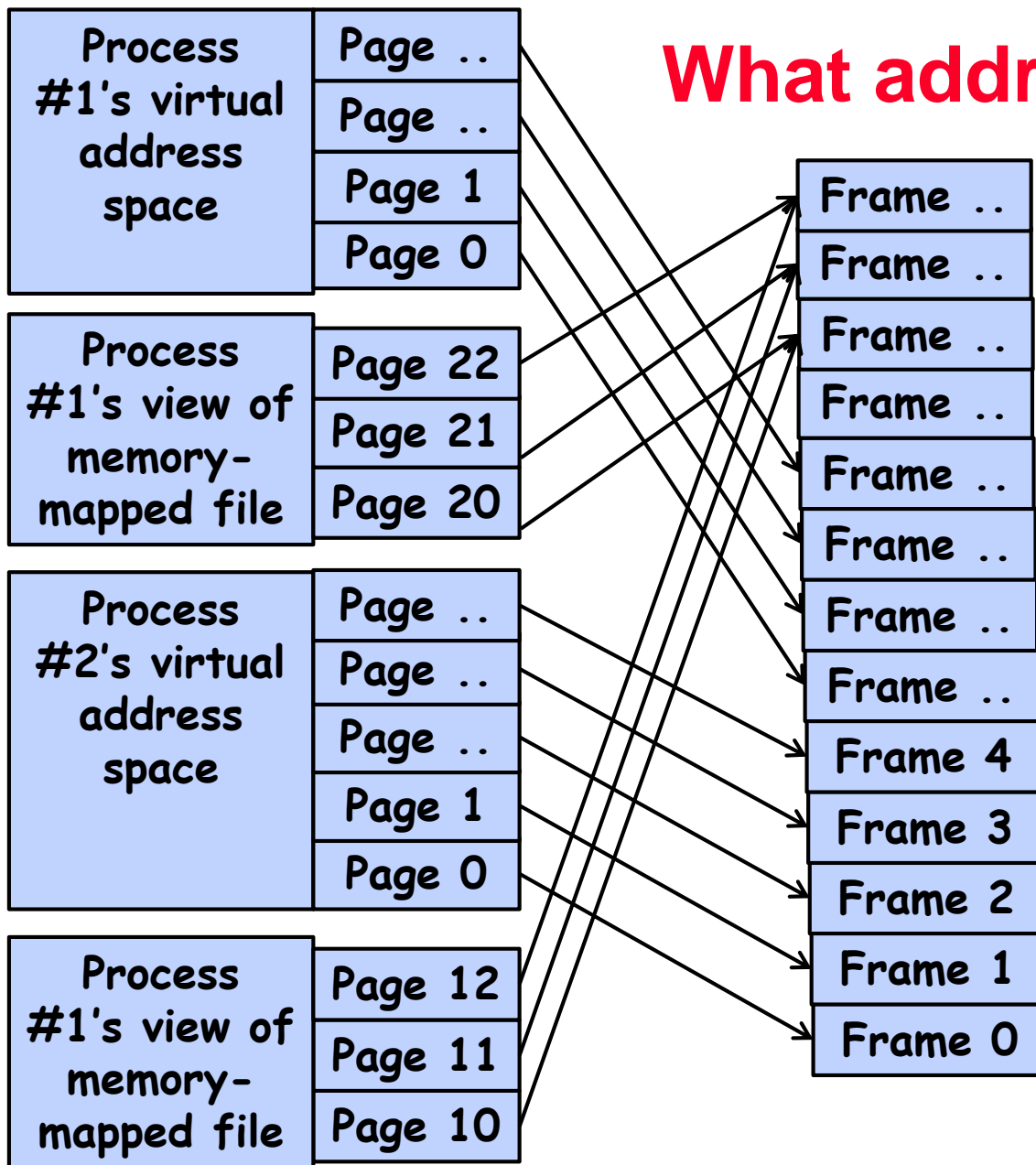
- Two different processes sharing the same code (read only)

What address translation is for



- When virtual pages are initially allocated, they all share the same physical page, initialised to zero
- When a write occurs, a page fault results in a fresh writable page being allocated

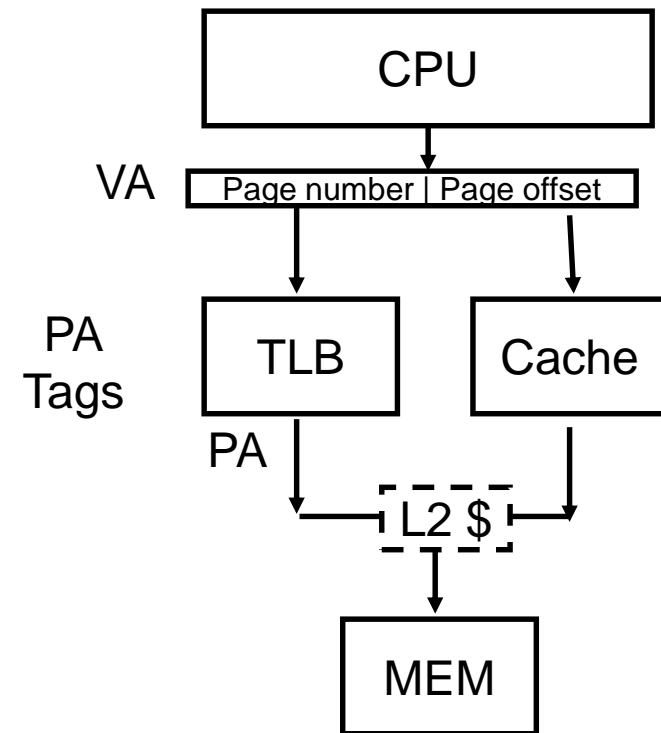
What address translation is for



- Two different processes sharing the same memory-mapped file (with both having read and write access permissions)

2. Fast Cache Hits by Avoiding Translation: Index with Physical Portion of Address

- If index is physical part of address, can start tag access in parallel with translation so that can compare to physical tag
- Limits cache to page size: what if want bigger caches and still use same trick?
 - Higher associativity
 - Page coloring
 - A cache conflict occurs if two cache blocks that have the same tag (physical address) are mapped to two different virtual addresses
 - Make sure OS never creates a page table mapping with this property



2. Fast Cache Hits by Avoiding Translation: Index with Physical Portion of Address

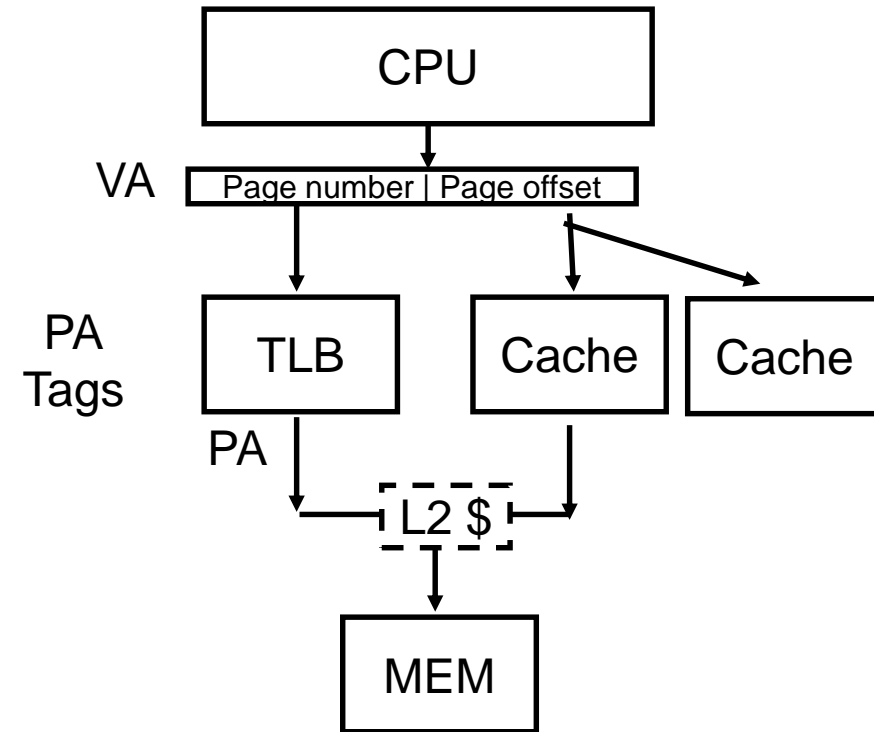
- If index is physical part of address, can start tag access in parallel with translation so that can compare to physical tag
- Limits cache to page size: what if want bigger caches and still use same trick?

- Option 1: Higher associativity

- This is an attractive and common choice
- Consequence: L1 caches are often highly associative

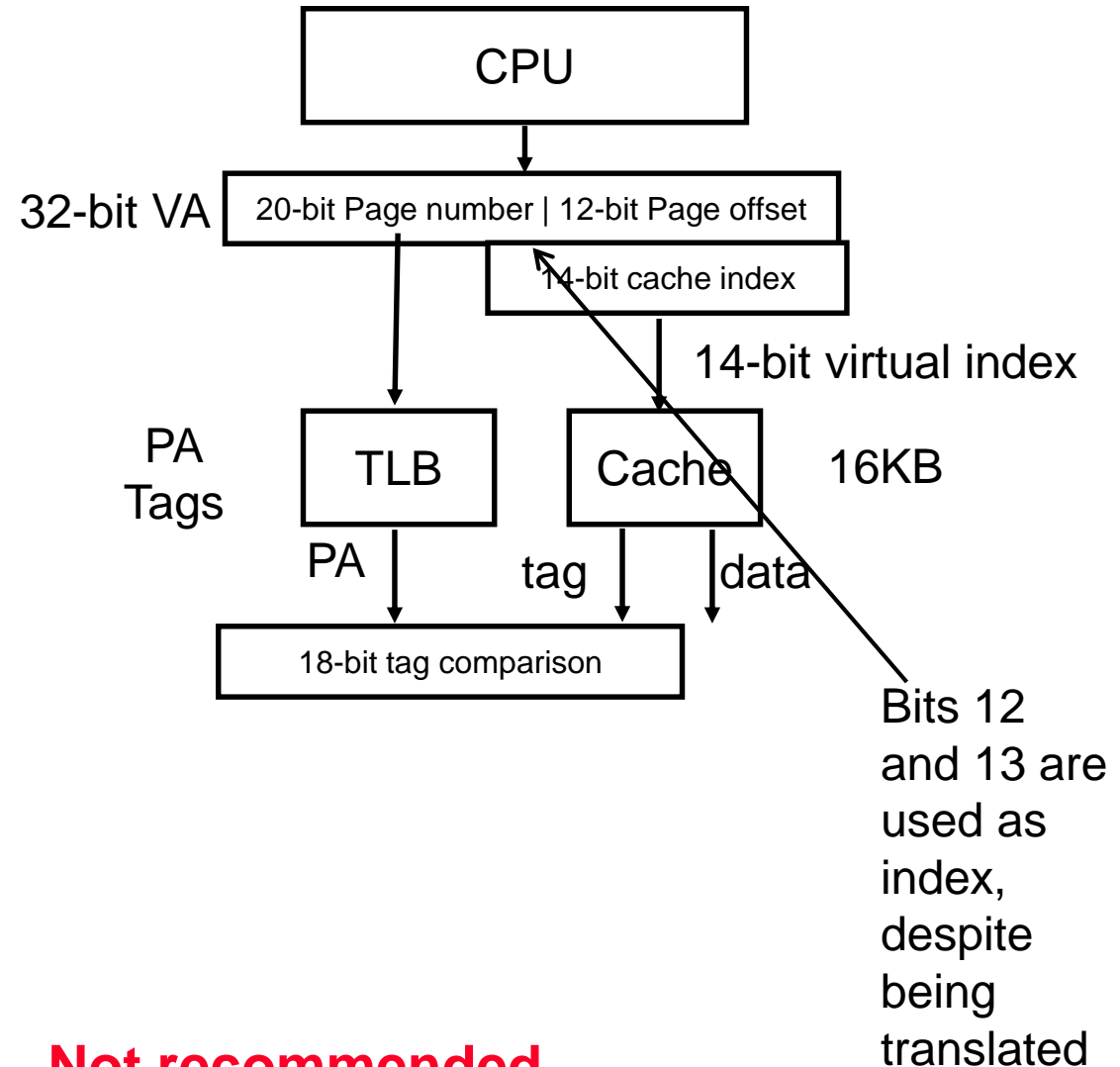
- Option 2: Page coloring

- Get the operating system to help – see next slide



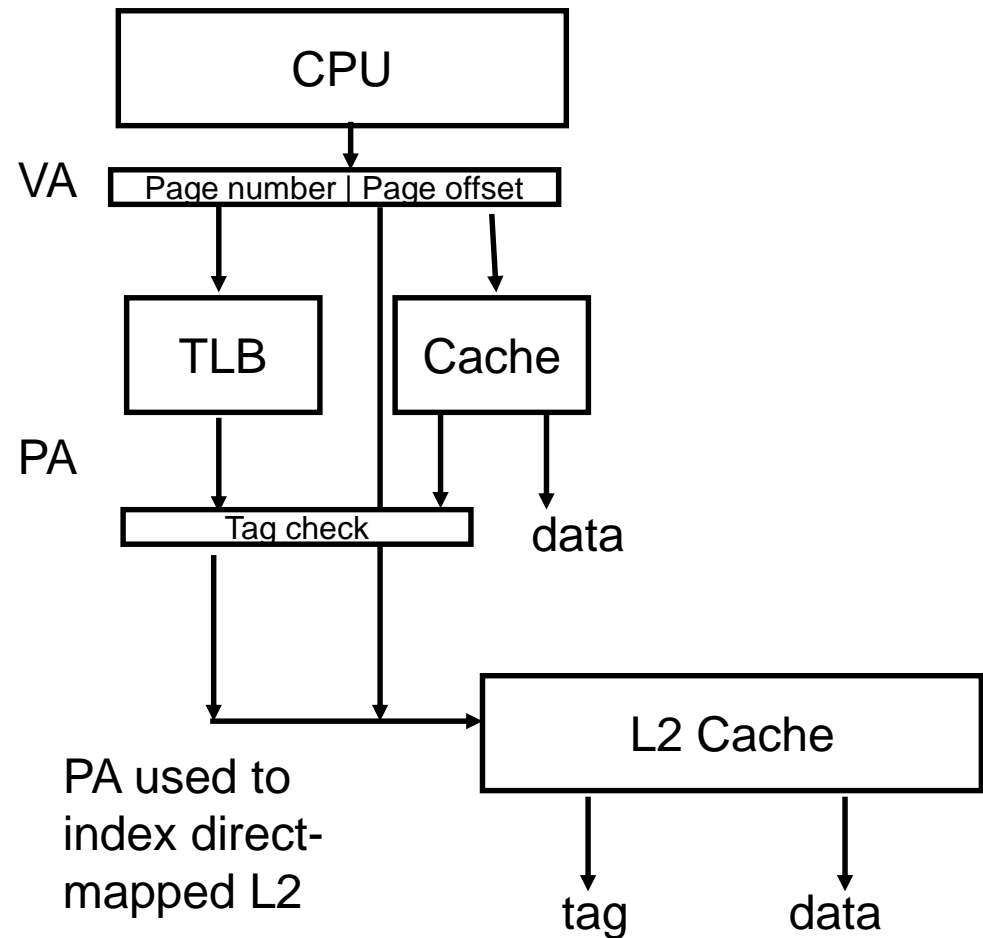
What if you *insist* on using some translated bits as index bits?

- Page colouring for **synonym consistency**:
- “A cache conflict occurs if two cache blocks that have the same tag (physical address) are mapped to two different virtual addresses”
- So if the OS needs to create two virtual memory regions, A and B within the same process, mapping the **same** physical address region
 - So A[0] and B[0] have *different* VAs
 - But after translation refer to the same location
 - We need to ensure that the virtual addresses that we assign to A[0] and B[0] match in bits 12&13
 - So they map to the same location in the cache
 - So they have only one value!

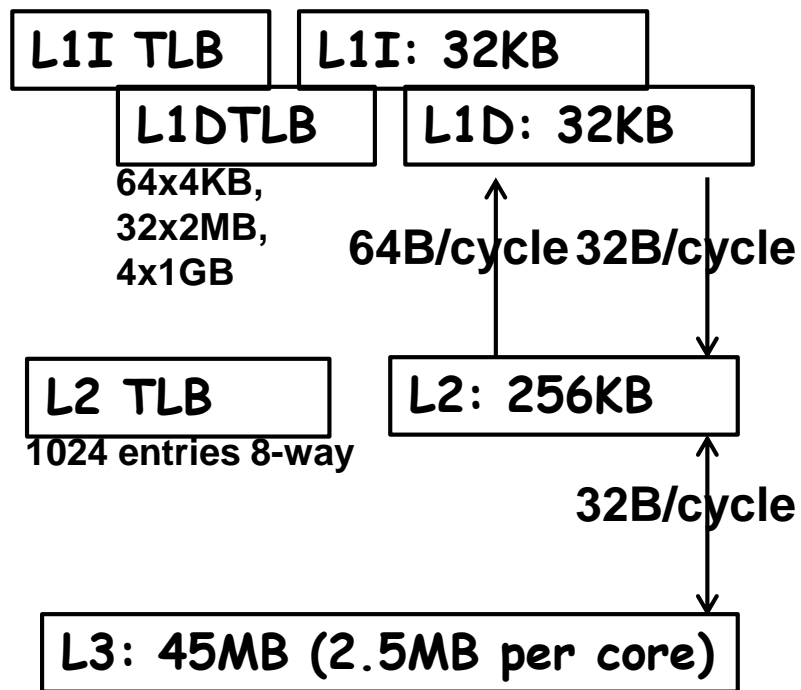


Associativity conflicts depend on address translation

- The L2 cache is indexed with *translated* address
- So the L2 associativity conflicts depend on the virtual-to-physical mapping
- It would be helpful if the OS could choose non-conflicting pages for frequently-accessed data!
(page colouring for conflict avoidance)
- Or at least, make sure adjacent pages don't map to the same L2 index



TLBs in Haswell



L1: 32KB, 8-way associative I and D

L1D: writeback, two 256-bit loads and a 256-bit store every cycle

So each L1 way is $32/8=4\text{KB}$

Virtually indexed, Physically Tagged (VIPT)

L2 and L3 are physically indexed

TLBs support three different page sizes – 4KB, 2MB, 1GB

● **Example: Intel Haswell e5 2600 v3**

3: Fast Hits by pipelining Cache

Case Study: MIPS R4000

● 8 Stage Pipeline:

- IF—first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
- IS—second half of access to instruction cache.
- RF—instruction decode and register fetch, hazard checking and also instruction cache hit detection.
- EX—execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
- DF—data fetch, first half of access to data cache.
- DS—second half of access to data cache.
- TC—tag check, determine whether the data cache access hit.
- WB—write back for loads and register-register operations.

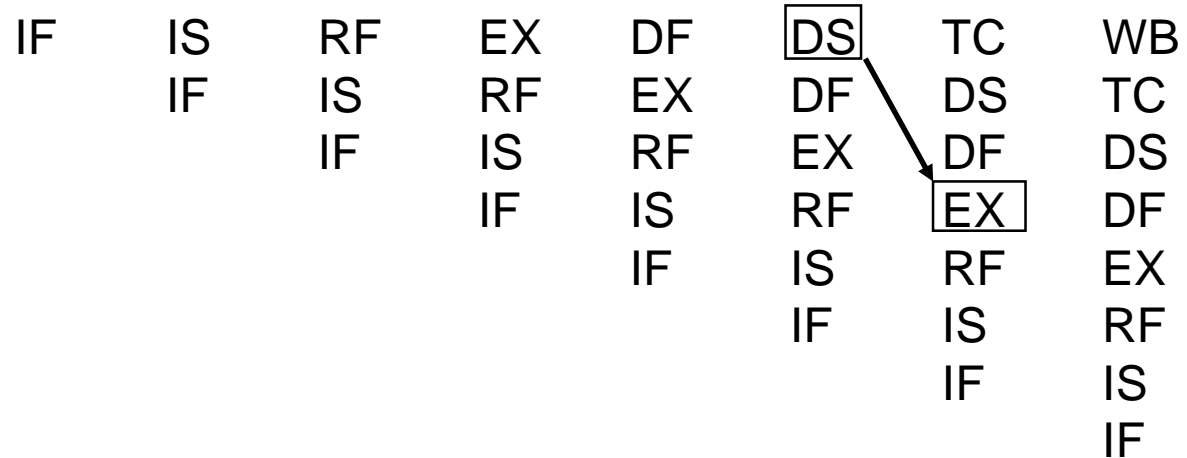
● What is impact on Load delay?

- Need 2 instructions between a load and its use!

- **Pipeline the cache**
 - **Eg one cache access per cycle, but with a 2-cycle access *latency***
- **What if we want to support multiple parallel accesses to the cache?**
 - **Divide the cache into several banks**
 - **Map addresses to banks in some way (low-order bits? Hash function?)**
 - **(see textbook) (Q: does this reduce AMAT?)**

Case Study: MIPS R4000

TWO Cycle Load Latency

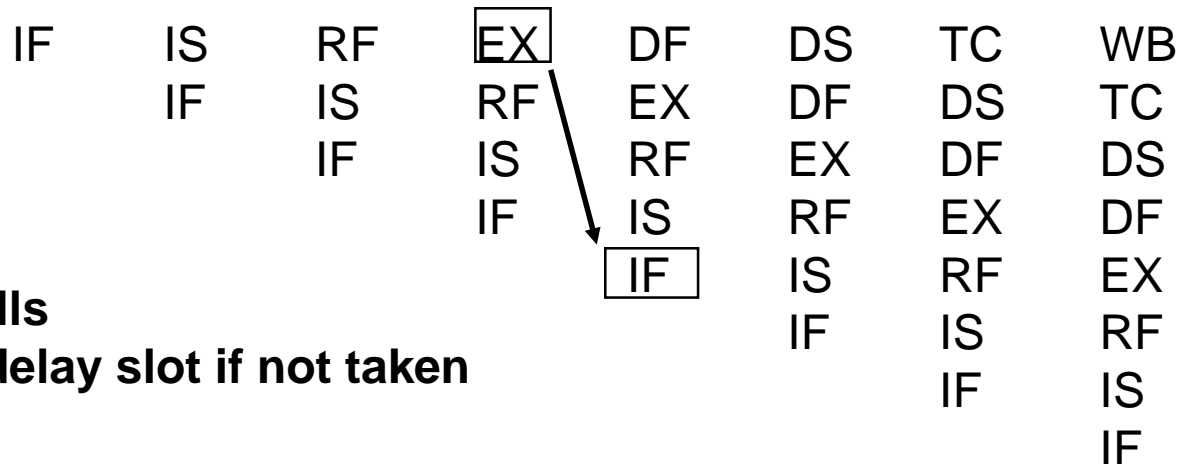


THREE Cycle Branch Latency

(conditions evaluated
during EX phase)

Delay slot plus two stalls

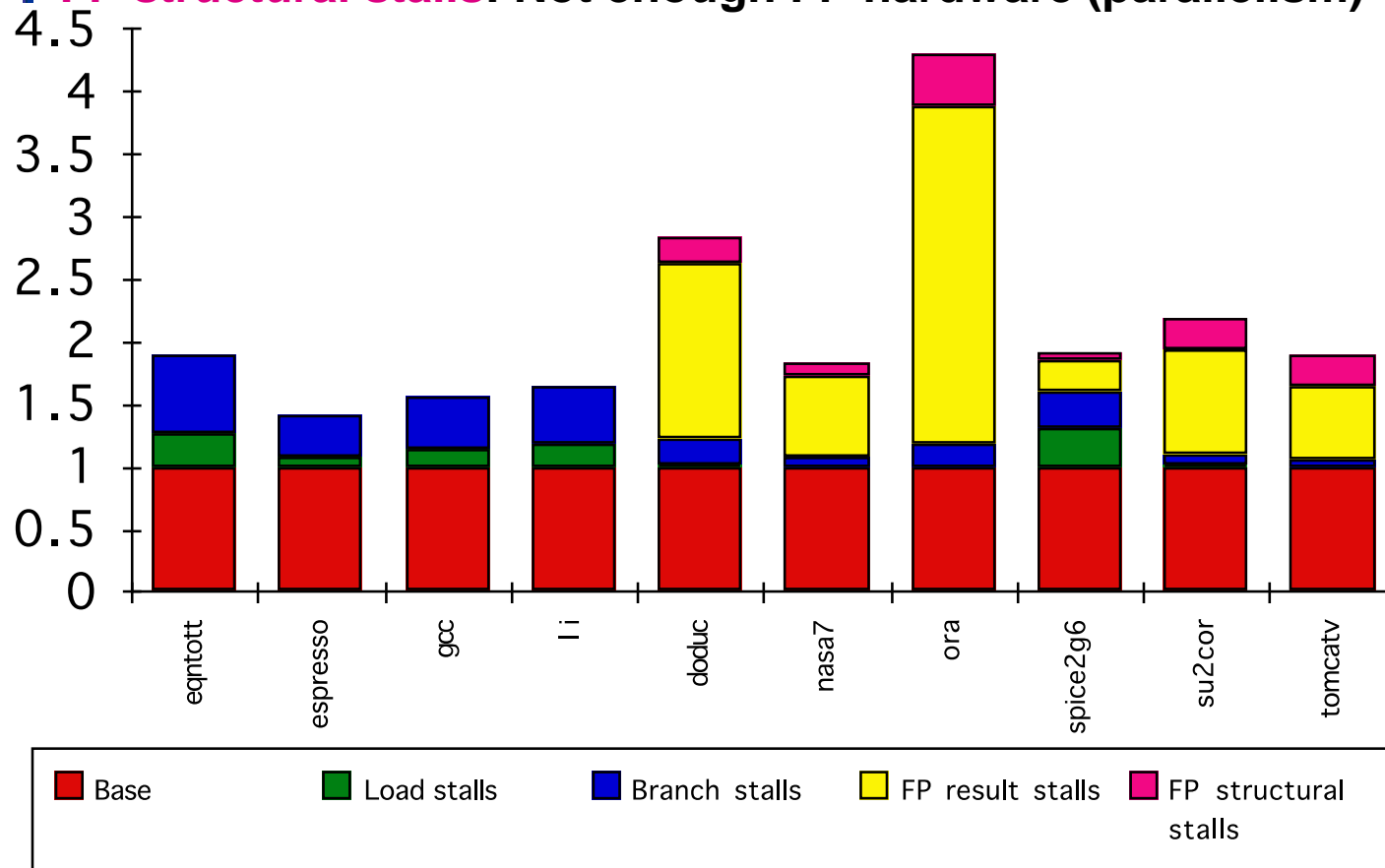
Branch likely cancels delay slot if not taken



R4000 Performance

- Not ideal CPI of 1:

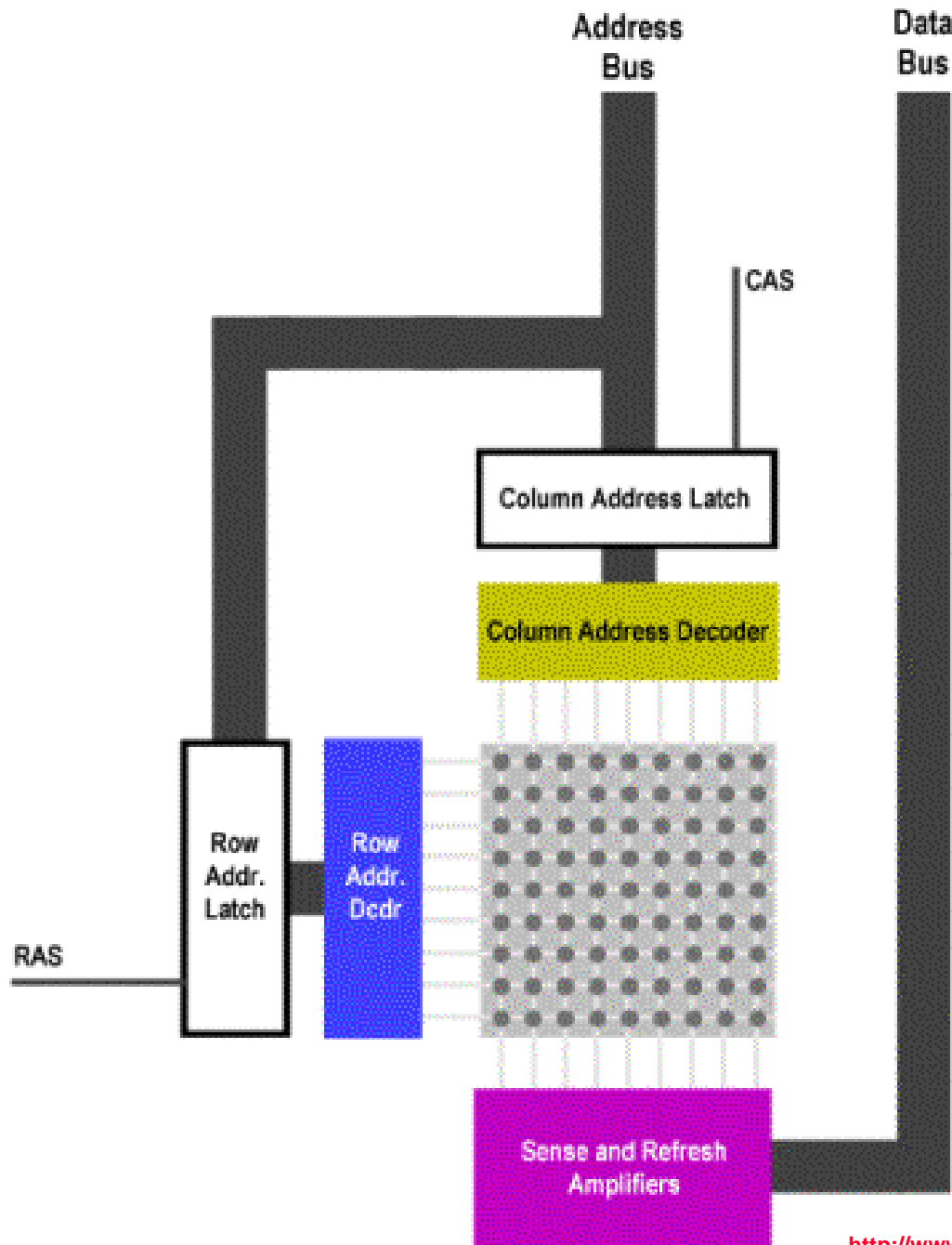
- **Load stalls** (1 or 2 clock cycles)
- **Branch stalls** (2 cycles + unfilled slots)
- **FP result stalls**: RAW data hazard (latency)
- **FP structural stalls**: Not enough FP hardware (parallelism)



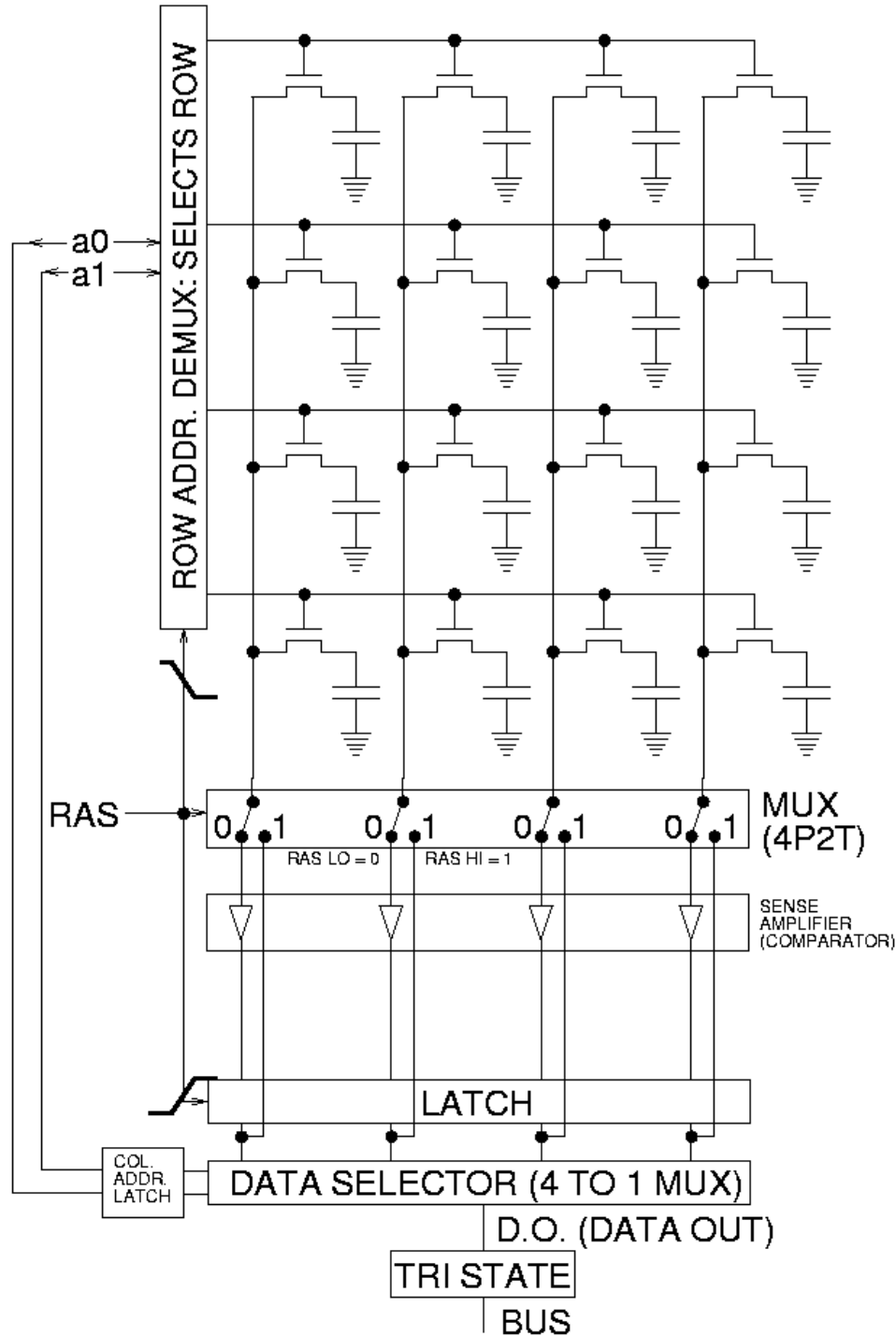
Cache Optimization Summary

	<i>Technique</i>	<i>MR</i>	<i>MP</i>	<i>HT</i>	<i>Complexity</i>
miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Prefetching of Instr/Data	+			2
	Compiler Controlled Prefetching	+			3
	Compiler Reduce Misses	+			0
miss penalty	Priority to Read Misses		+		1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2

DRAM array design



- Square array of cells
- Address split into Row address and Column Address bits
- Row address selects row of cells to be activated
- Cells discharge
- Cell state latched by per-column sense amplifiers
- Column address selects data for output
- Data must be written back to selected row



DRAM array design

- Square array of cells
- Address split into Row address and Column Address bits
- Row address selects row of cells to be activated
- Cells discharge
- Cell state latched by per-column sense amplifiers
- Column address selects data for output
- Data must be written back to selected row

DRAM cell design

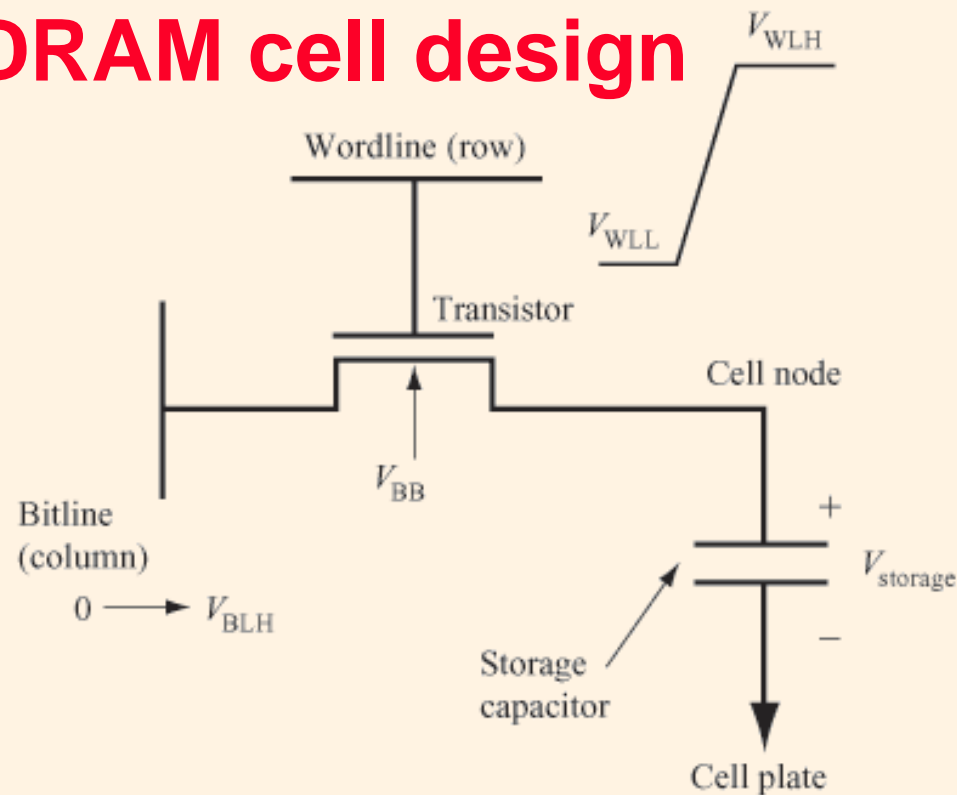


Figure 1

Schematic of a one-transistor DRAM cell [1]. The array device (transistor) is addressed by switching the wordline voltage from V_{WLL} (wordline-low) to V_{WLH} (wordline-high), enabling the bitline and the capacitor to exchange charge. In this example, a data state of either a “0” (0 V) or a “1” (V_{BLH}) is written from the bitline to the storage capacitor. V_{BB} is the electrical bias applied to the p-well.

<http://www.research.ibm.com/journal/rd/462/mandelman.html>

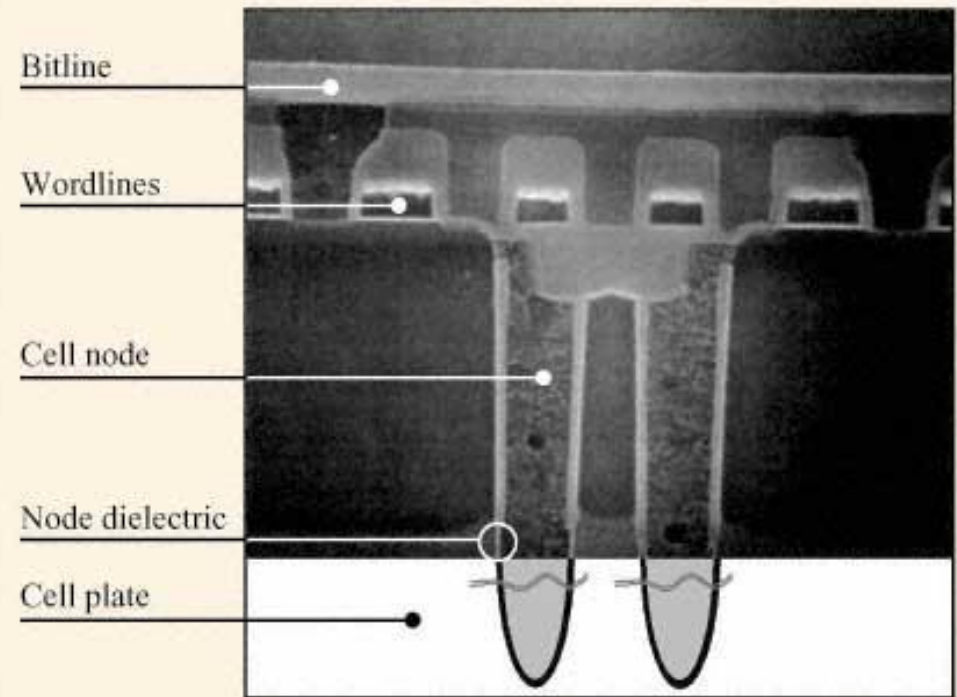


Figure 4

SEM photomicrograph of 0.25-μm trench DRAM cell suitable for scaling to 0.15 μm and below. Reprinted with permission from [17]; © 1995 IEEE.

- Single transistor
- Capacitor stores charge
- Decays with time
- Destructive read-out

Main Memory Background

● Performance of Main Memory:

● Latency: Cache Miss Penalty

- **Access Time**: time between request and word arrives
- **Cycle Time**: time between requests

● Bandwidth: I/O & Large Block Miss Penalty (L2)

● Main Memory is **DRAM**: Dynamic Random Access Memory

- Dynamic since needs to be **refreshed** periodically (8 ms, 1% time)
- Addresses divided into 2 halves (Memory as a 2D matrix):
 - **RAS** or **Row Access Strobe**
 - **CAS** or **Column Access Strobe**

● Cache (at least L1,L2) uses **SRAM**: Static Random Access Memory

- No refresh (6 transistors/bit vs. 1 transistor)
- **Size**: DRAM is *4-8 times smaller than SRAM*,
Cost/Cycle time: SRAM is *8-16 times faster than DRAM*

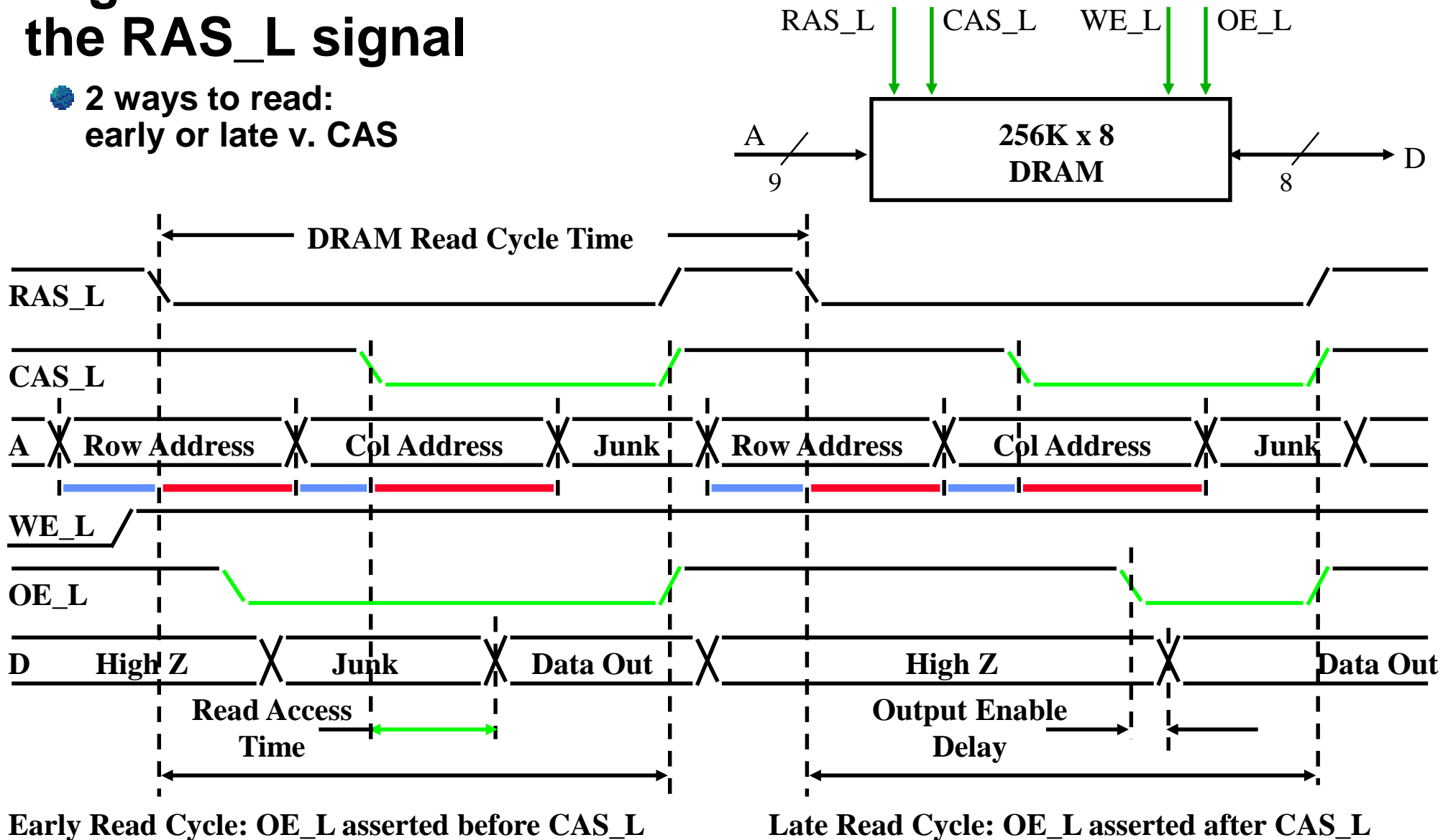
4 Key DRAM Timing Parameters

- **t_{RAC}** : minimum time from RAS line falling to the valid data output.
 - Quoted as the speed of a DRAM when you buy it
 - A typical 4Mb DRAM $t_{\text{RAC}} = 60 \text{ ns}$
- **t_{RC}** : minimum time from the start of one row access to the start of the next.
 - $t_{\text{RC}} = 110 \text{ ns}$ for a 4Mbit DRAM with a t_{RAC} of 60 ns
- **t_{CAC}** : minimum time from CAS line falling to valid data output.
 - 15 ns for a 4Mbit DRAM with a t_{RAC} of 60 ns
- **t_{PC}** : minimum time from the start of one column access to the start of the next.
 - 35 ns for a 4Mbit DRAM with a t_{RAC} of 60 ns

Every DRAM access begins at assertion of the RAS_L signal

- 2 ways to read:
early or late v. CAS

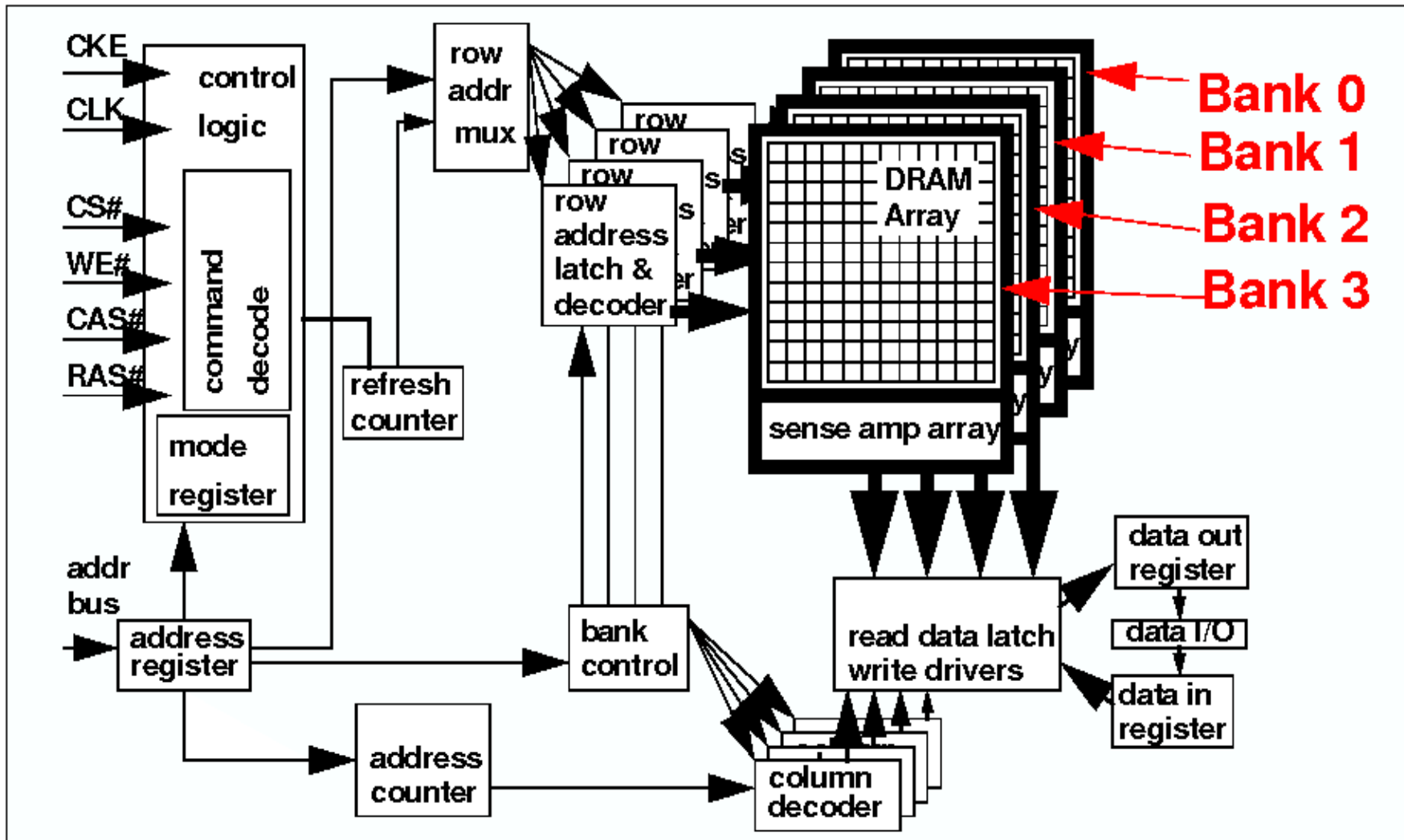
DRAM Read Timing



Early Read Cycle: OE_L asserted before CAS_L

Late Read Cycle: OE_L asserted after CAS_L

Putting it all together...



- **Architecture of SDRAM chip (based on Micron MT48LC32M4A2 data sheet)**
- From David Taiwei Tang's PhD thesis (<https://www.ece.umd.edu/~blj/papers/thesis-PhD-wang--DRAM.pdf>)

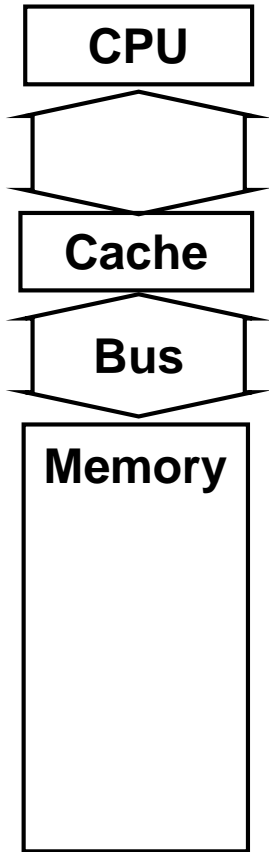
DRAM topics we don't have time for...

- Refresh
- Reliability, the “rowhammer” attack, mitigation
- DRAM energy
- Stacking, eg Micron's Hybrid Memory Cube (HMC)
- Processing-in-memory, near-memory processing
- Bulk copy and zeroing (easy intra subarray, trickier inter) – Rowclone. Cache coherence?
- Non-volatile memory (NVM), storage-class memory (SCM), Phase-change memory (PCM), Flash, Crosspoint
- Why NVM isn't a filesystem

More cache stuff:

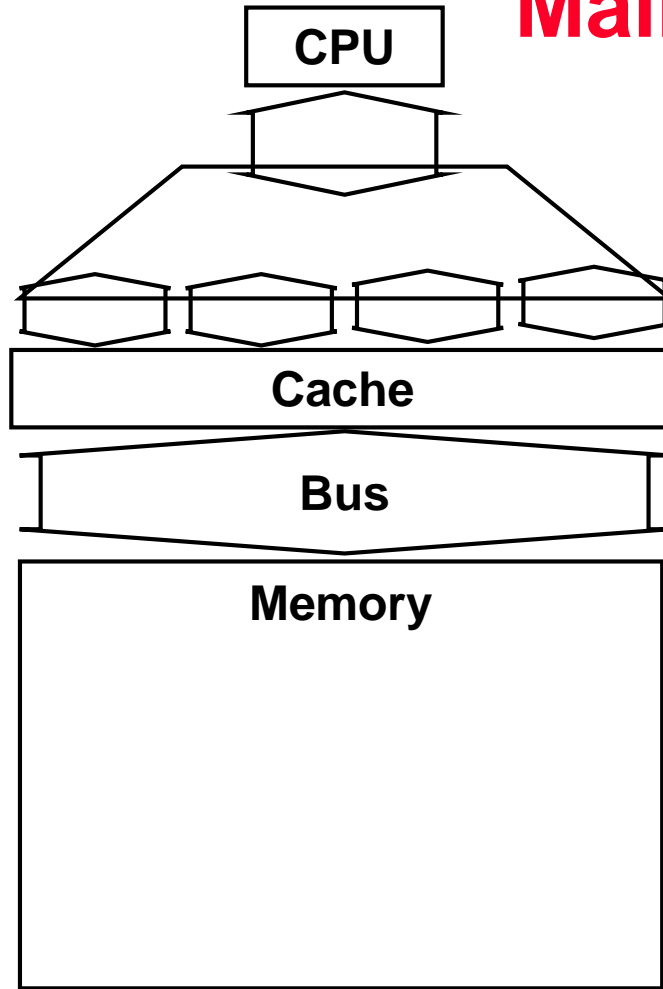
- Compressed skewed caches
- Energy optimisations in the memory hierarchy
- Content locality, upper bit content locality

Main Memory Organizations



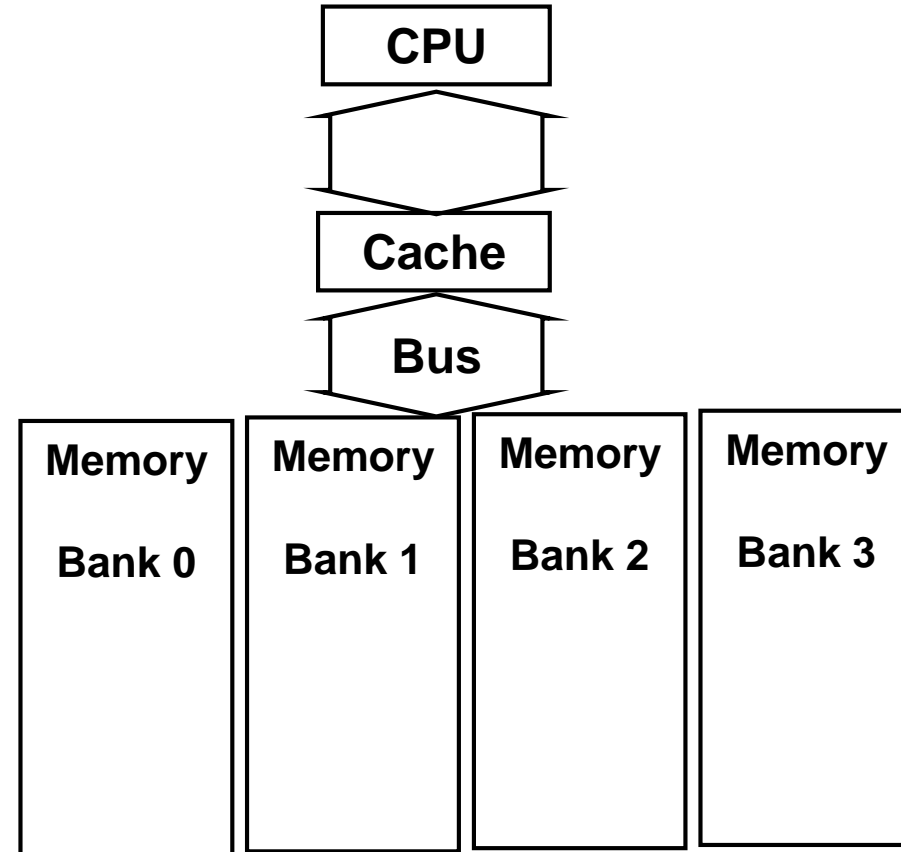
● Simple:

- CPU, Cache, Bus, Memory same width (32 or 64 bits)



● Wide:

- CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits; UltraSPARC 512)



● Interleaved:

- CPU, Cache, Bus 1 word; Memory N Modules (4 Modules); example is *word interleaved*

Avoiding Bank Conflicts

- Lots of banks

```
int x[256][512];  
for (j = 0; j < 512; j = j+1)  
    for (i = 0; i < 256; i = i+1)  
        x[i][j] = 2 * x[i][j];
```

- Conflicts occur even with 128 banks, since 512 is multiple of 128, conflict on word accesses
- SW: loop interchange or declaring array not power of 2 (“array padding”)
- HW: Prime number of banks
 - bank number = address mod number of banks
 - address within bank = address / number of words in bank
 - modulo & divide per memory access with prime no. banks?
 - address within bank = address mod number words in bank
 - bank number? easy if 2^N words per bank

- **Motivation:**
 - Failures/time *proportional* to number of bits!
 - As DRAM cells shrink, more vulnerable
- **Went through period in which failure rate was low enough without error correction that people didn't do correction**
 - DRAM banks too large now
 - Servers always use corrected memory systems
- **Basic idea: add redundancy through parity/ECC bits**
 - Simple but wasteful version:
 - Keep three copies of everything, vote to find right value
 - 200% overhead, so not good!
 - Common configuration: Random error correction
 - SEC-DED (single error correct, double error detect)
 - One example: 64 data bits + 8 parity bits (11% overhead)
 - Really want to handle failures of physical components as well
 - Organization is multiple DRAMs/SIMM, multiple SIMMs
 - Want to recover from failed DRAM and failed SIMM!
 - Requires more redundancy to do this
 - All major vendors thinking about this in high-end machines

Main Memory Summary

- **DRAM arrays have separate row address latency**
- **DRAM reads are destructive – needs to be written back**
- **Memory parallelism:**
 - **DRAM packages typically include multiple DRAM arrays, which can be row-addressed in parallel**
 - **Wider Memory – increase transfer bandwidth to transfer up to a whole row (or more?) in a single transaction**
 - **Interleaved Memory: allowing multiple addresses to be accessed in parallel**
- **Bank conflicts occur when two different rows in the same DRAM array are required at the same time**
 - **Hardware strategies, software strategies...**
- **Need error correction (which costs in space and time)**
- **Non-volatile memory technologies – fast-moving field**
 - **Flash – requires block erasure, suffers burnout**
 - **Phase-change – slow writes, bitwise addressable**

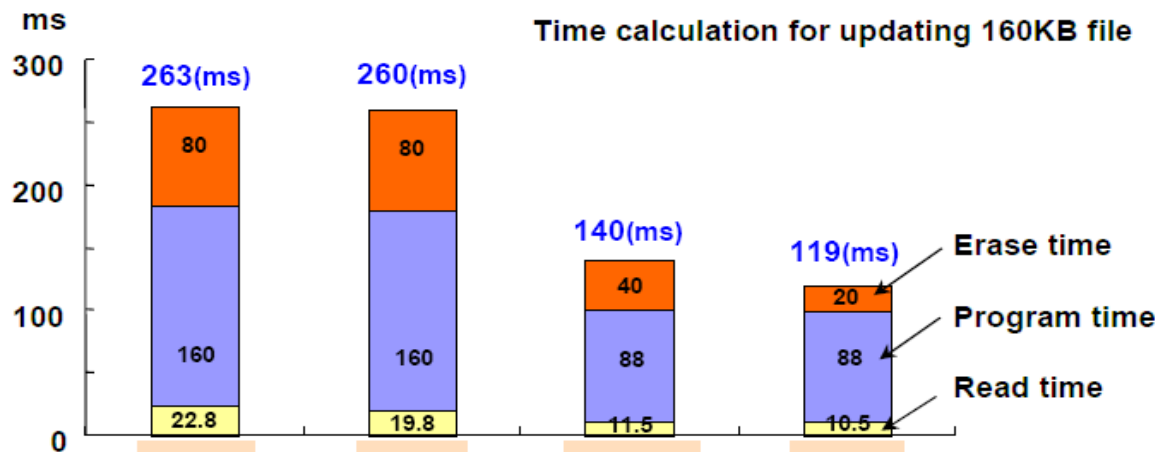
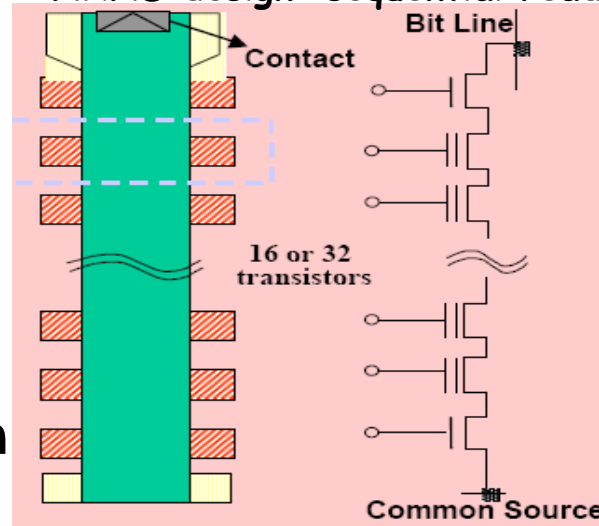
Extra material for interest

FLASH

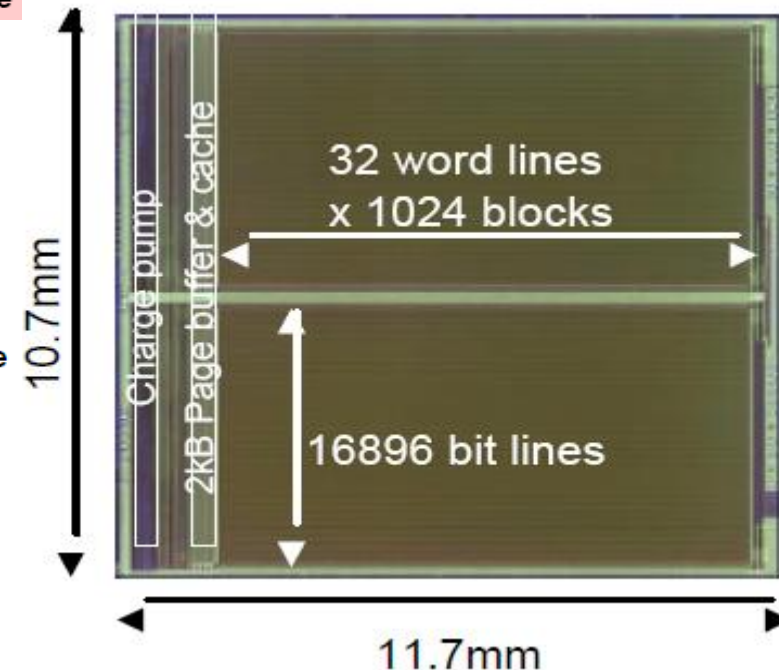
More esoteric Storage Technologies?

- Mosfet cell with two gates
- One “floating”
- To program, charge tunnels via $<7\text{nm}$ dielectric
- Cells can only be erased (reset to 0) in blocks

NAND design: sequential read, high density



	8Mb	16Mb	32/64Mb	128Mb
Page size	256Byte	256Byte	512Byte	512Byte
Block size	4KByte	4KByte	8KByte	16KByte
Read/page	10us	10us	10/7us	10us
Program/page	250us	250us	250/200us	200us
Erase/block	2ms	2ms	2ms	2ms

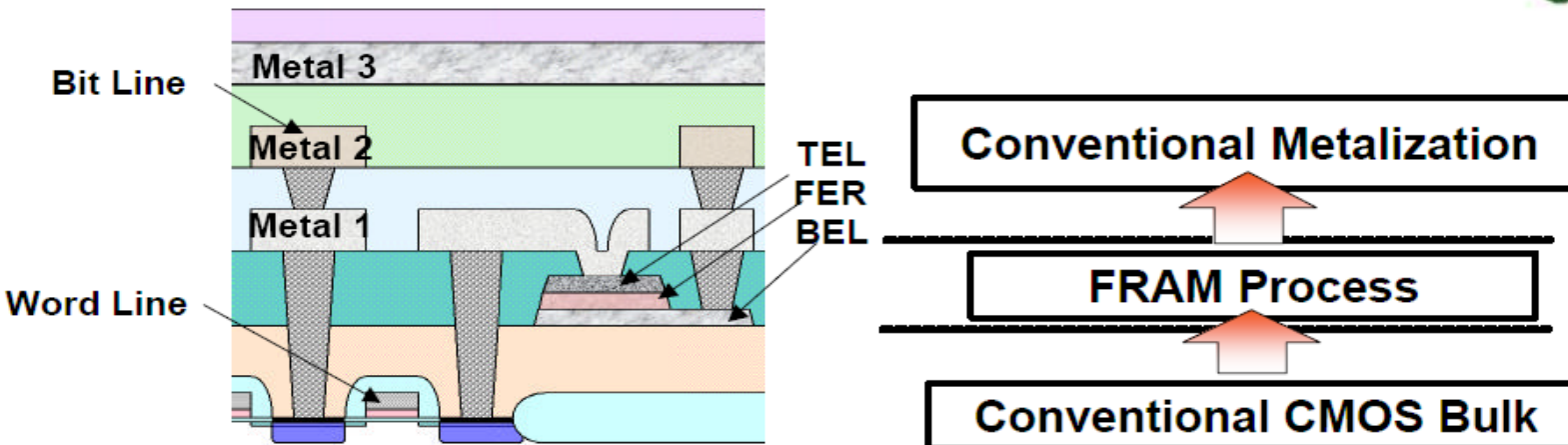
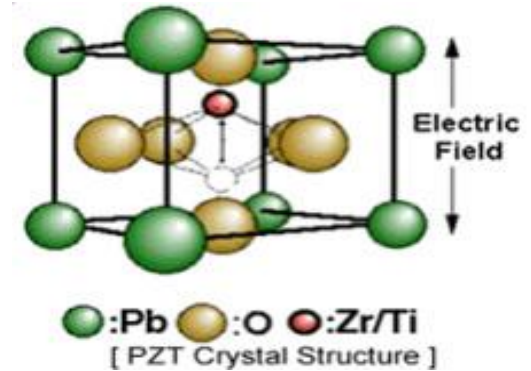


1 Gbit NAND Flash memory

Diverse non-volatile memory technologies

● FRAM

- Perovskite ferroelectric crystal forms dielectric in capacitor, stores bit via phase change
- 100ns read, 100ns write
- Very low write energy (ca.1nJ)



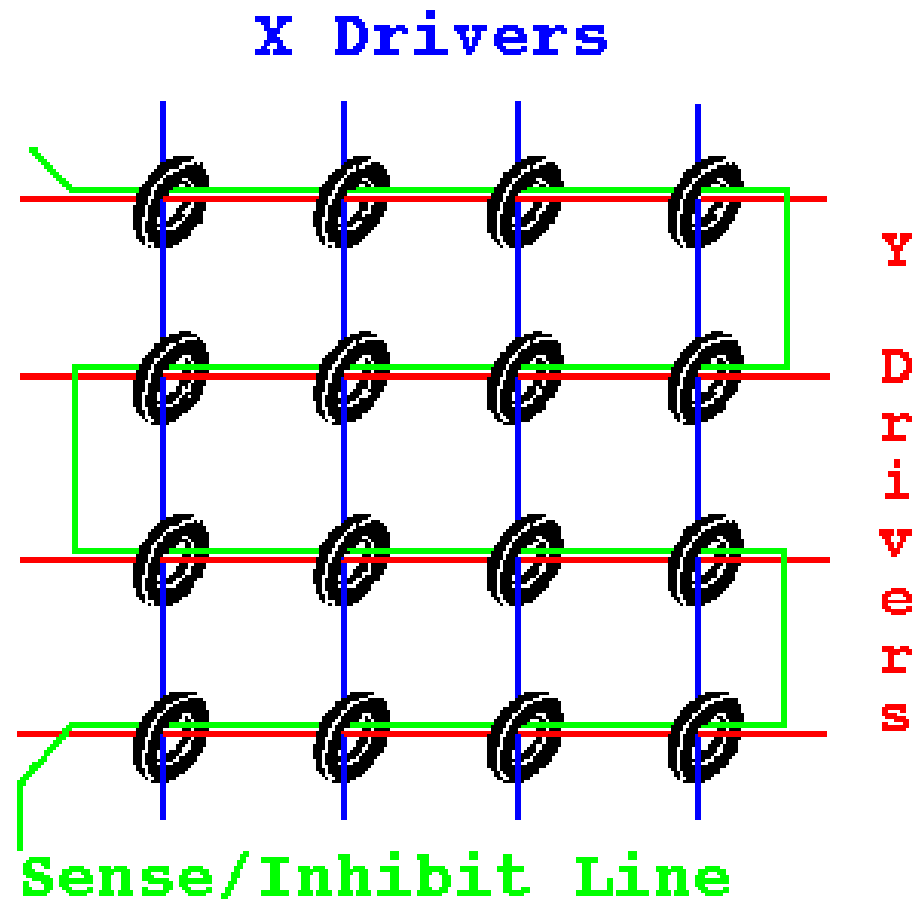
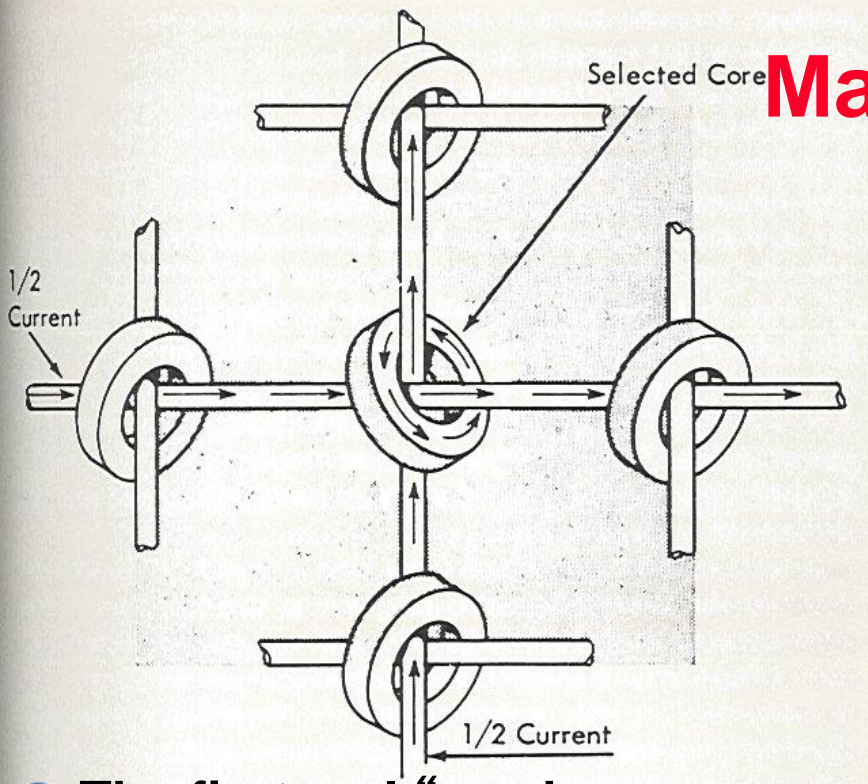
Additional FRAM process
between conventional CMOS bulk and metalization



Compatible with conventional CMOS technology
and existing CMOS cell libraries

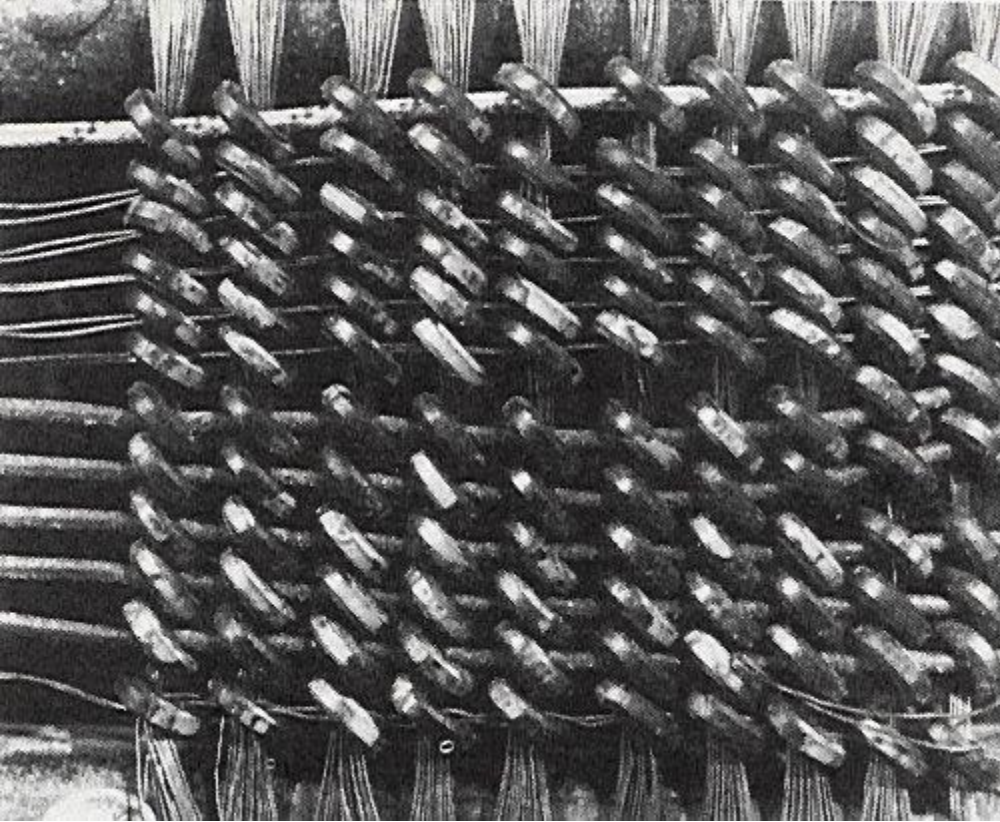
- Fully integrated with logic fab process
- Currently used in Smartcards/RFID
- Soon to overtake Flash?
- See also phase change RAM

Main Memory Deep Background



Pulse on sense line if any core flips its magnetisation state

- The first real “random-access memory” technology was based on magnetic “cores” – tiny ferrite rings threaded with copper wires
- That’s why people talk about “Out-of-Core”, “In-Core,” “Core Dump”
- Non-volatile, magnetic
- Lost out when 4 Kbit DRAM became available
- Access time 750 ns, cycle time 1500-3000 ns



- The first magnetic core memory, from the **IBM 405 Alphabetical Accounting Machine**. The photo shows the single drive lines through the cores in the long direction and fifty turns in the short direction. The cores are 150 mil inside diameter, 240 mil outside, 45 mil high. This experimental system was tested successfully in April 1952.

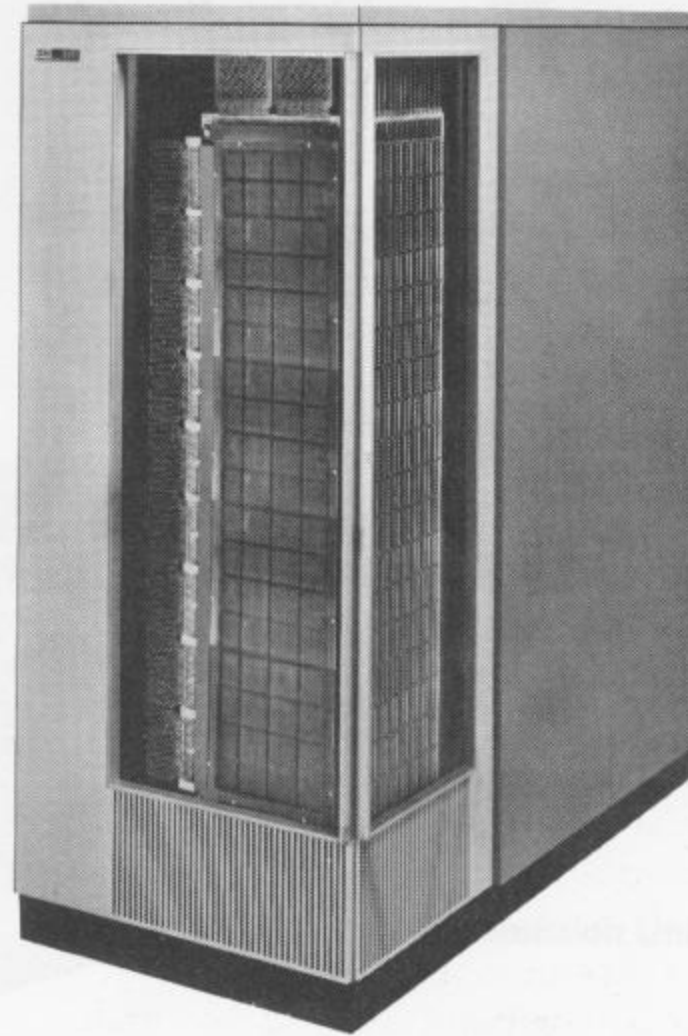
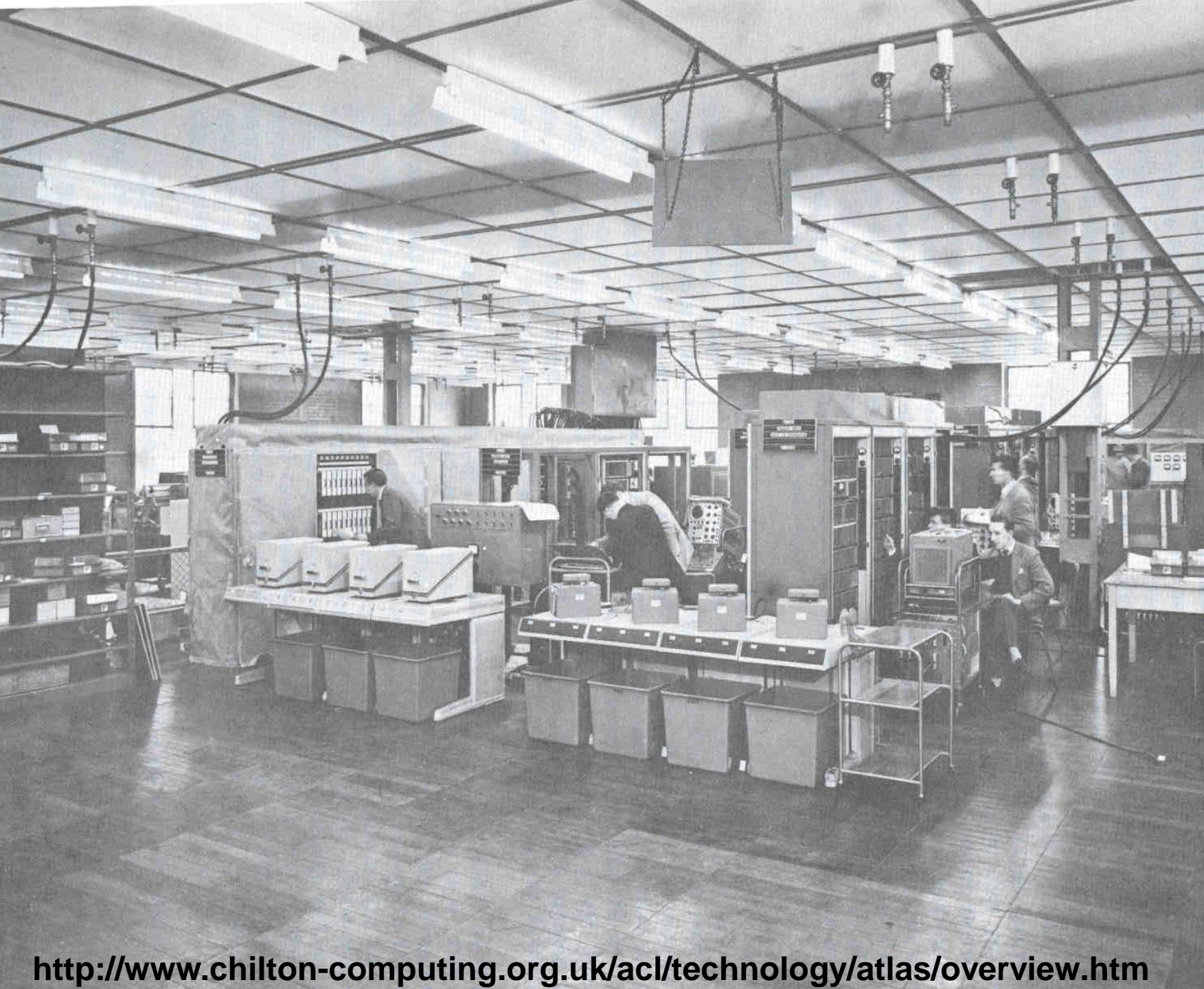


Figure 10. IBM 2361 Core Storage

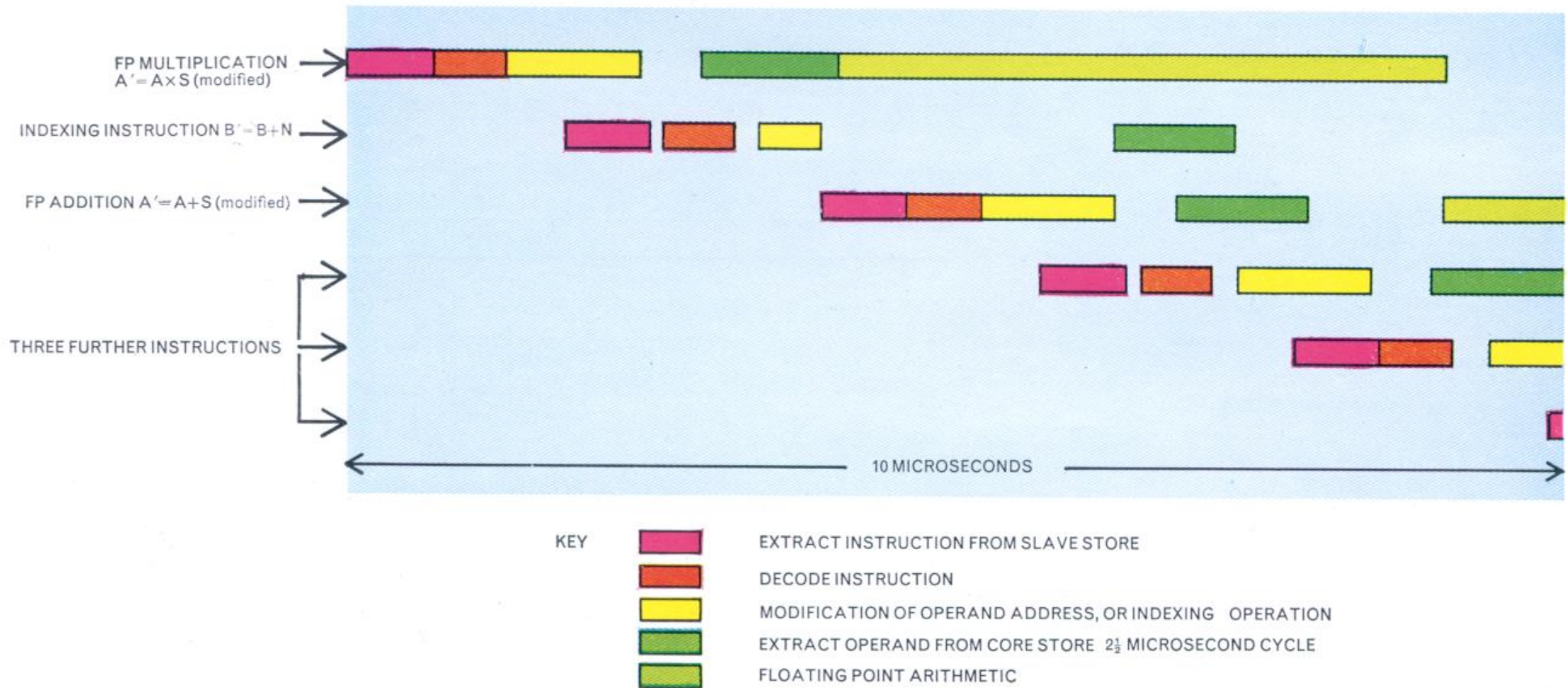
- 524,000 36-bit words and a total cycle time of eight microseconds in each memory (1964 – for the IBM7094)



● Atlas

- First was at University of Manchester
- University of London had the second one
- Commissioned May 1964
- Shut down Sept 1972

Pipelined instruction processing in Atlas



<http://www.chilton-computing.org.uk/acl/technology/atlas/overview.htm>

- Atlas is most famous for pioneering virtual memory
- Also
 - Pipelined execution
 - Cache memory ("slave store") – 32 words
 - Floating point arithmetic hardware