ECE 5730
Memory Systems
Spring 2009

Cache Content Management

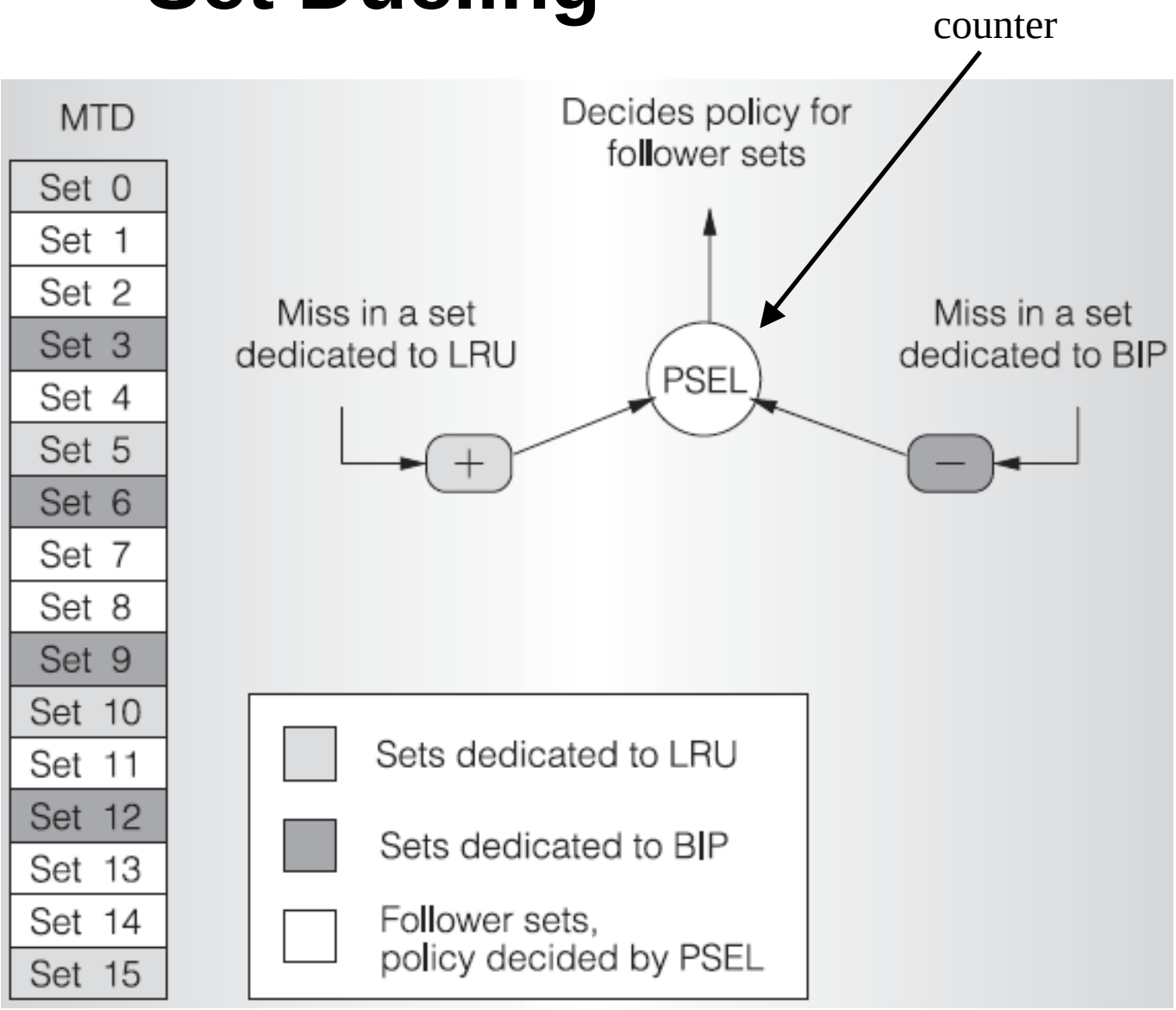
Cornell University

Announcements

Errata

Set Dueling

- **Some sets use LRU, others BIP**
- ***Follower sets* follow the policy that does best**



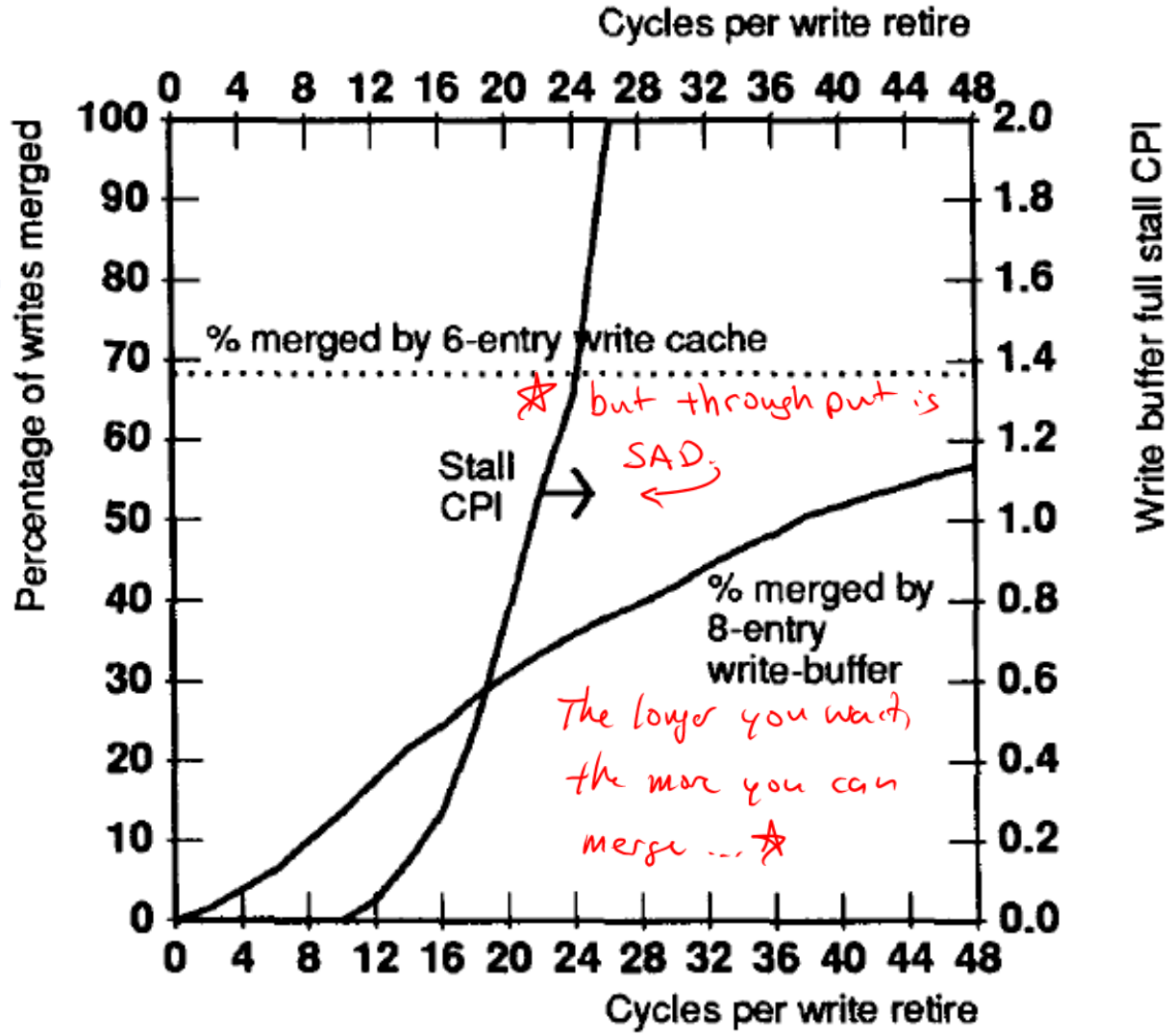
[Qureshi08]

- no discussion in paper of layout of LRU/BIP sets.
- still, obviously laid out to combat power of 2 strides

Errata

When to Empty the Write Buffer?

- Want to retain data to increase the merging of data into a wider L2 write
- Want to empty the buffer to prevent it from getting full and stalling the pipeline

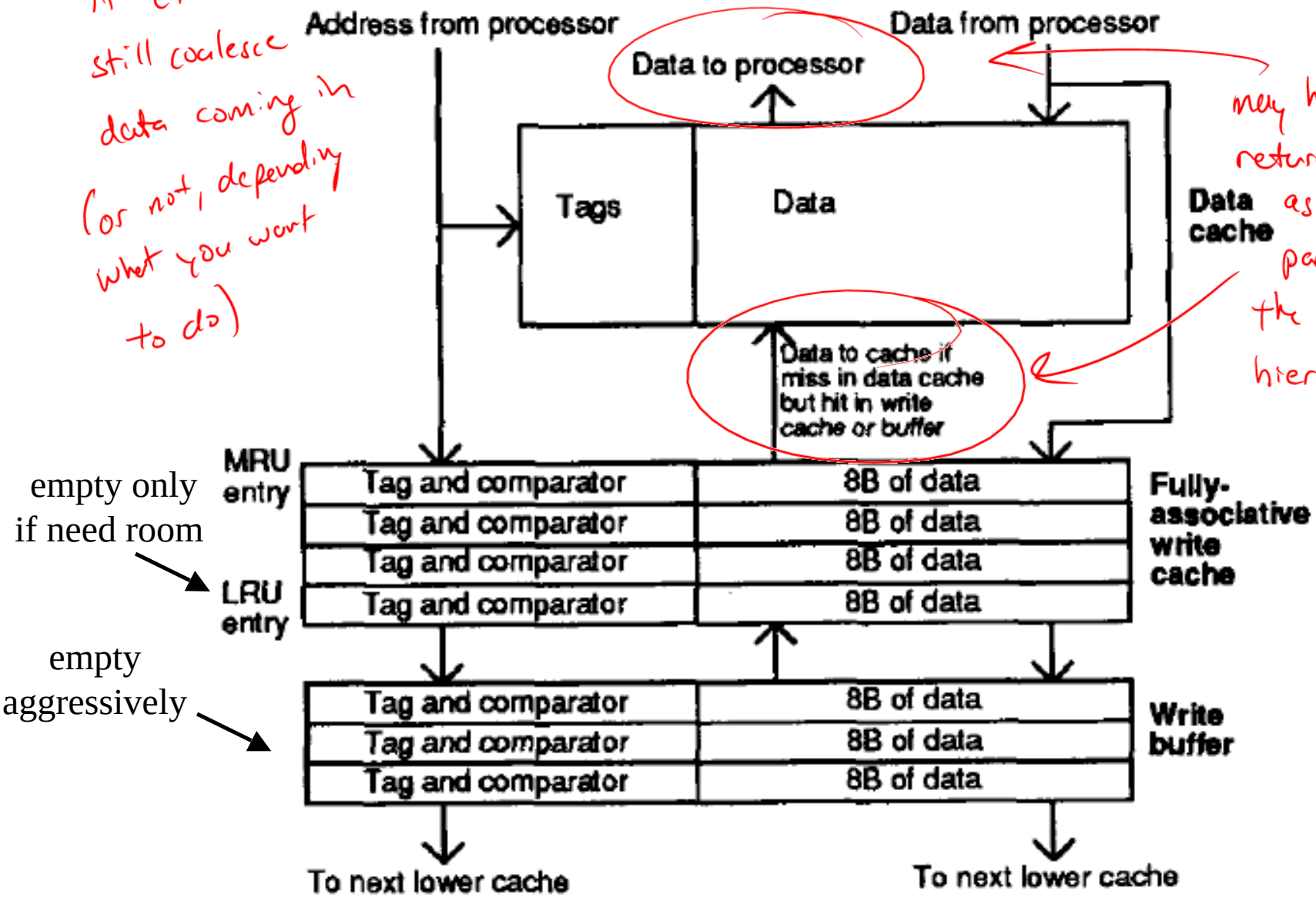


[Jouppi93]

Errata

Write Cache

it can still coalesce data coming in (or not, depending what you want to do)



empty only if need room

empty aggressively

[Jouppi93]

Cache Content Management

- **Partitioning heuristics**
  - **Which items get placed and where (cache level, cache sets/ways, buffers)**
- **Fetching heuristics**
  - **When to bring an item into the cache**
- **Locality optimizations**
  - **Change layout or ordering to improve reuse**

Decisions can be made at runtime (*on-line heuristics*),
at design/compile time (*off-line heuristics*),
or a combination of the two (*combined approaches*)

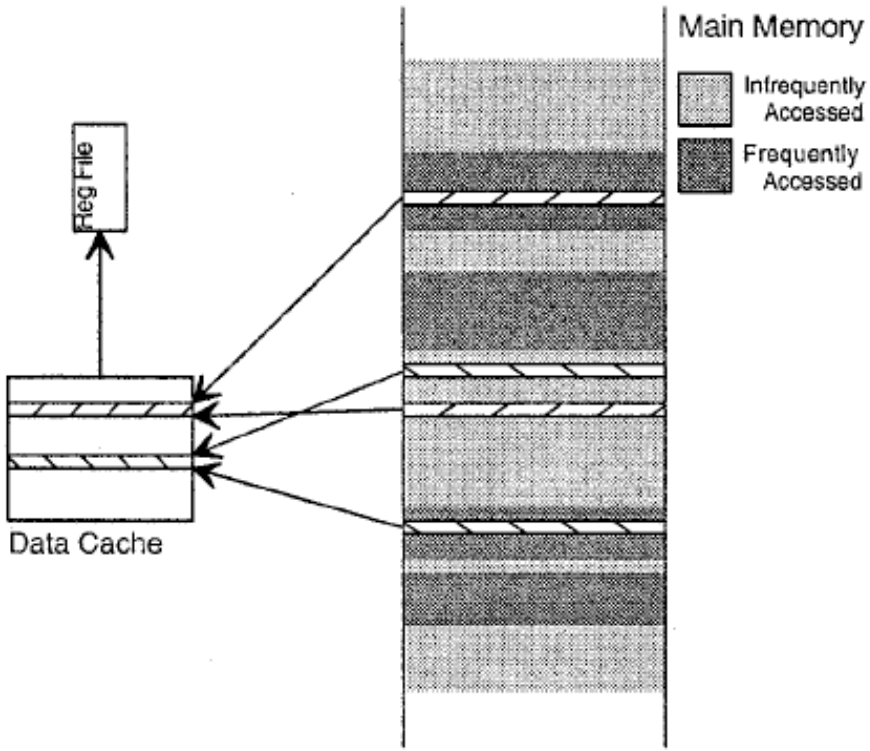
On-line Partitioning Heuristics

- Replacement policies
- Write policies
- Cache/not-cache strategies

Cache/Not-Cache Strategies

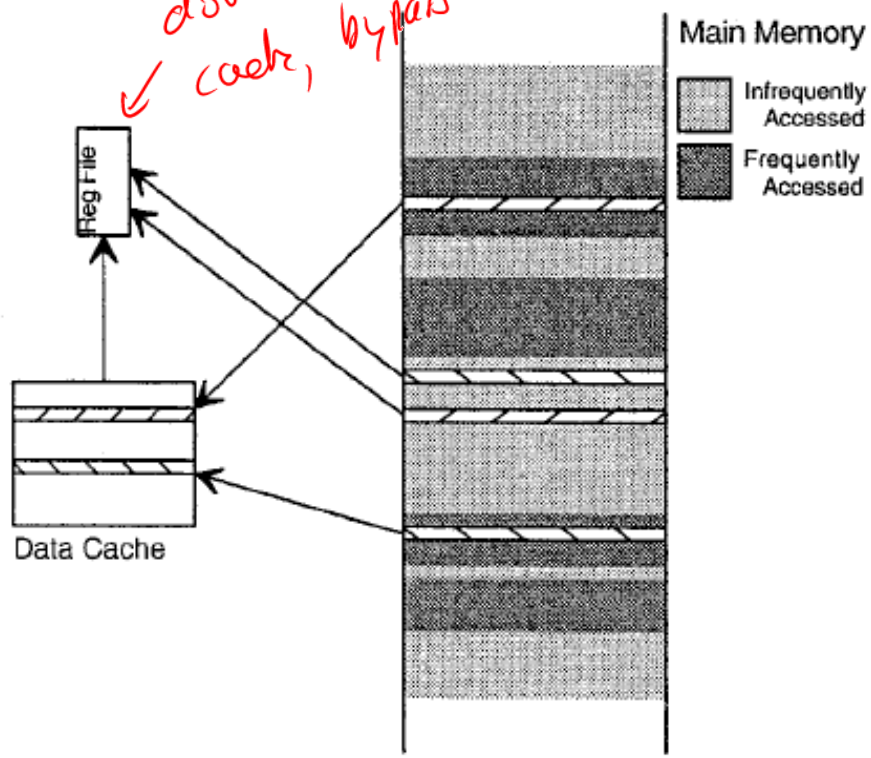
- Usual approach is to treat all data the same
- But all data does not behave the same way!
  - Some loads cause an exorbitant number of misses
  - Some data is reused more than other data
- Cache can dynamically decide what data to load and where to load it (e.g., L1 or L2)

An Example



conventional

don't store everything in the cache, bypass straight to regfile



selective bypassing

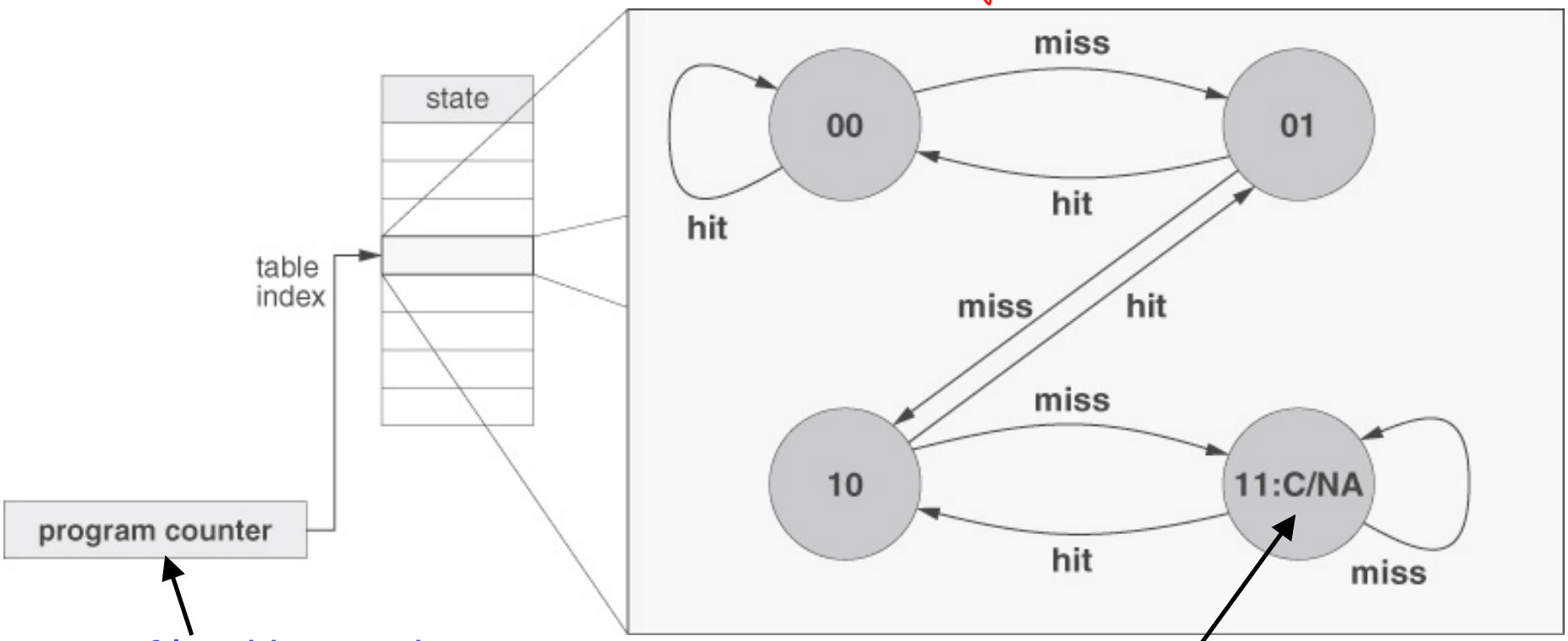
[Johnson97]

Miss-Driven Approach

- **Don't allocate space in the cache for blocks that miss too often**

← 2 in this case

n-bit saturating counter (# of bits is variable)



PC of load instruction used to index table of saturating counters

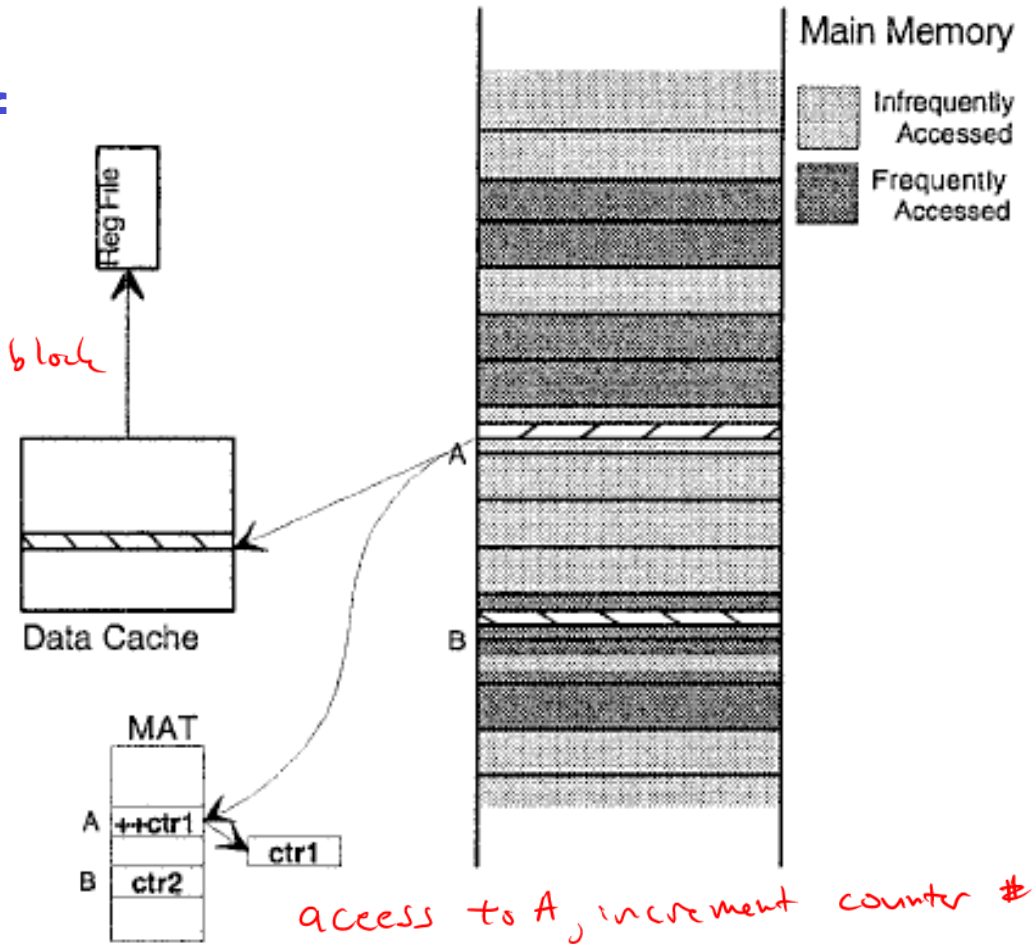
cacheable/non-allocatable

Frequency-Driven Approach

- Frequently accessed data is loaded into the cache
- Infrequently accessed data may bypass the cache (only placed in the register file)
- [Johnson97]

Memory Address Table (MAT)

- **Entries correspond to "macroblocks" of multiple cache blocks** look at a region instead of a single block
- **Counter in each entry indicates access frequency**
- **Counter (e.g., cntr1) is incremented on a MAT hit**



access to A, increment counter #1

access! not hit in cache

[Johnson97]

ctr 1 → A

ctr 2 → B

Memory Address Table (MAT)

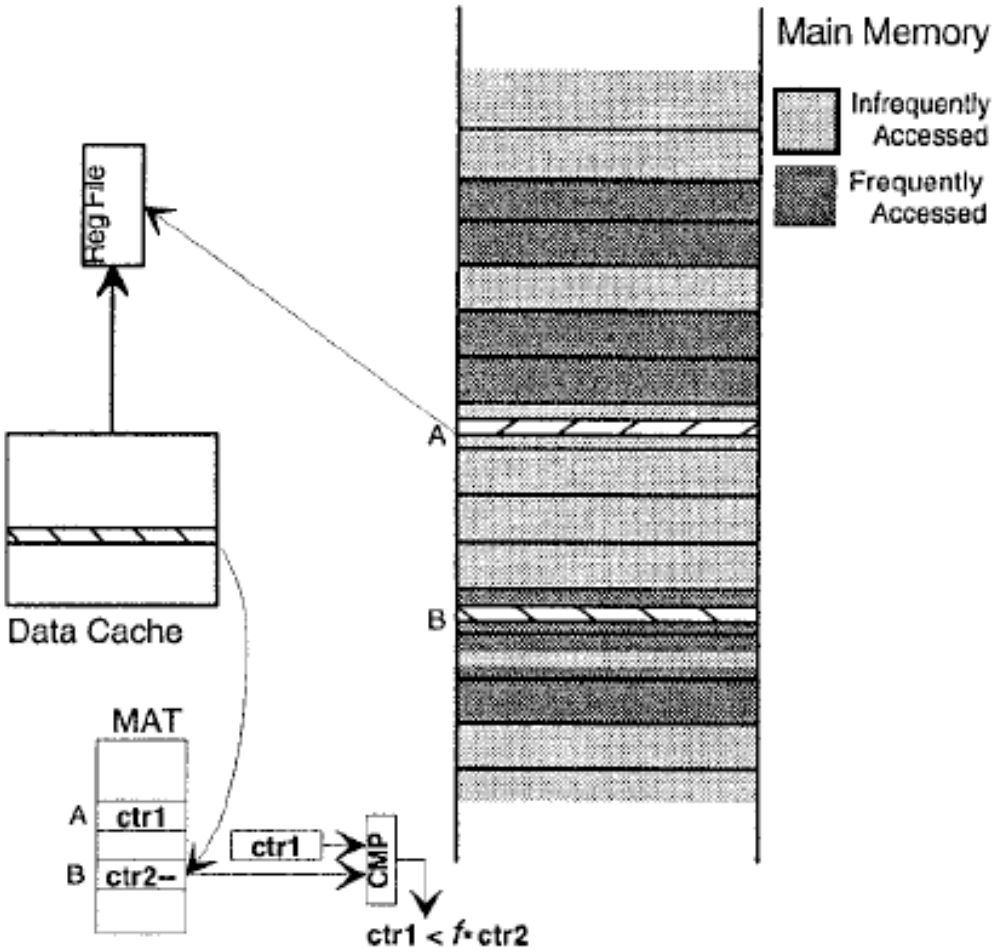
- **If a cache miss, counter (ctr2) for macroblock being replaced (B) compared with ctr1**

tuneable factor

- **If ctr1 < f•ctr2, bypass the cache**

→ if ctr2 is bigger than the other counters, don't evict B

- **ctr2 decremented, reflecting contention for that location**



[Johnson97]

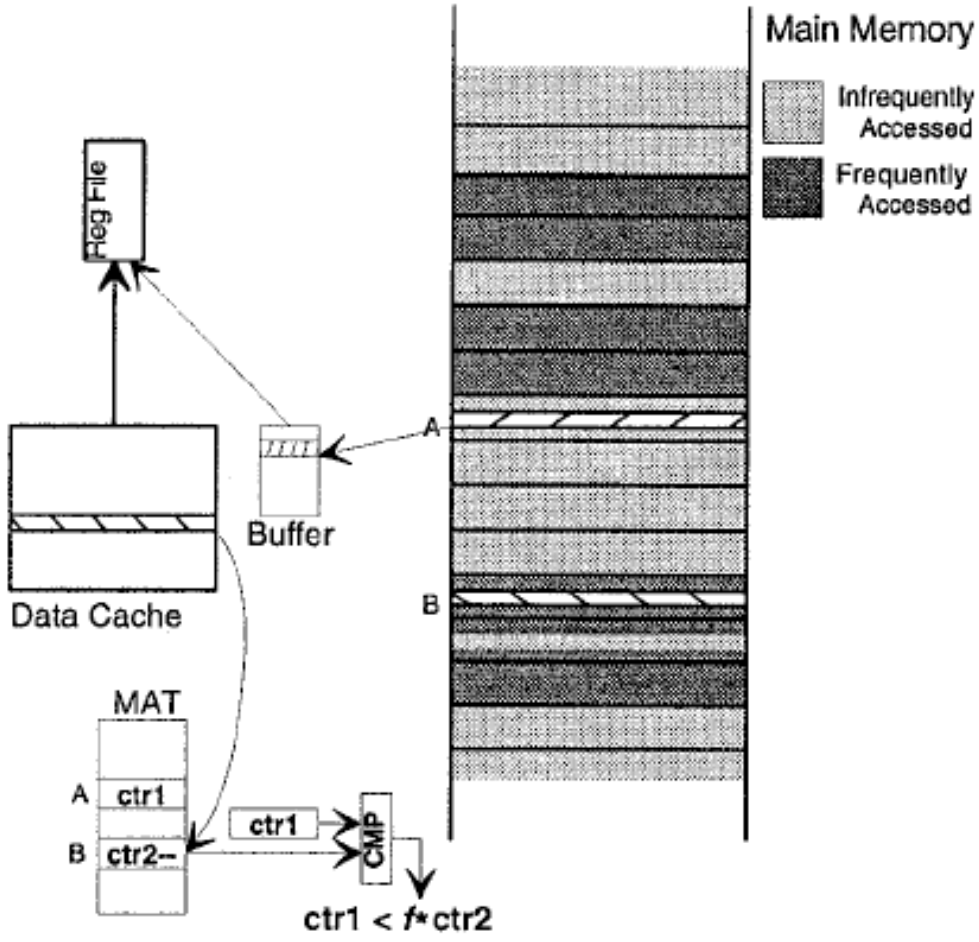
only bring stuff into the cache if we think it'll be accessed more. If not, just bypass to the reg file.

You can tweak it by messing w/ the counters

MAT + Bypass Buffer

- **Temporal locality of the bypassed data preserved with a small bypass buffer**
- **Operates like a victim cache**

The bypass goes to the bypass buffer instead of straight to the reg file



[Johnson97]

On-line Fetching Heuristics

- **Hardware that determines when to fetch instructions or data into the cache**

- **Demand fetch** standard operation
  - **Instructions/data brought into the cache on a miss**

- **Prefetching** makes more effective, sometimes => could hurt by consuming bandwidth / kicking stuff out of the cache
  - **Instructions/data are *predictively* accessed in advance of their being needed**

Simplistic Prefetching Approaches

here are 3 approaches!

- **Prefetch next block when current one accessed**

  fetch on every access! (if not already done)

- **Prefetch next block when current one misses**

  fetch on miss

- **Tagged prefetch**

  new bit, NOT the same as address tag

  - **Tag bit associated with each cache block**
  - **Initialized to 0 when block brought into the cache**
  - **Set to 1 when the block is accessed by the CPU**
  - **0 to 1 transition causes a prefetch of the next block**

  only prefetch on second access of the block

Generate too much downstream traffic, or don't prefetch early enough to hide modern memory latencies

problems with all of these approaches

Stream Buffers

called the "stream buffer"

by Norm Jouppi

→ does NOT fix the bandwidth issue

- Separate storage for prefetched lines

  → don't put prefetched stuff in the cache (don't pollute the cache)

  this is nice, but

- On a miss, successive cache lines prefetched in addition to fetching the requested line
  - Space is allocated in the SB but *available bit* = 0

    → allocate empty space

    sorta like the "valid" bit

- Prefetched lines are stored in the SB and become available (available bit = 1)

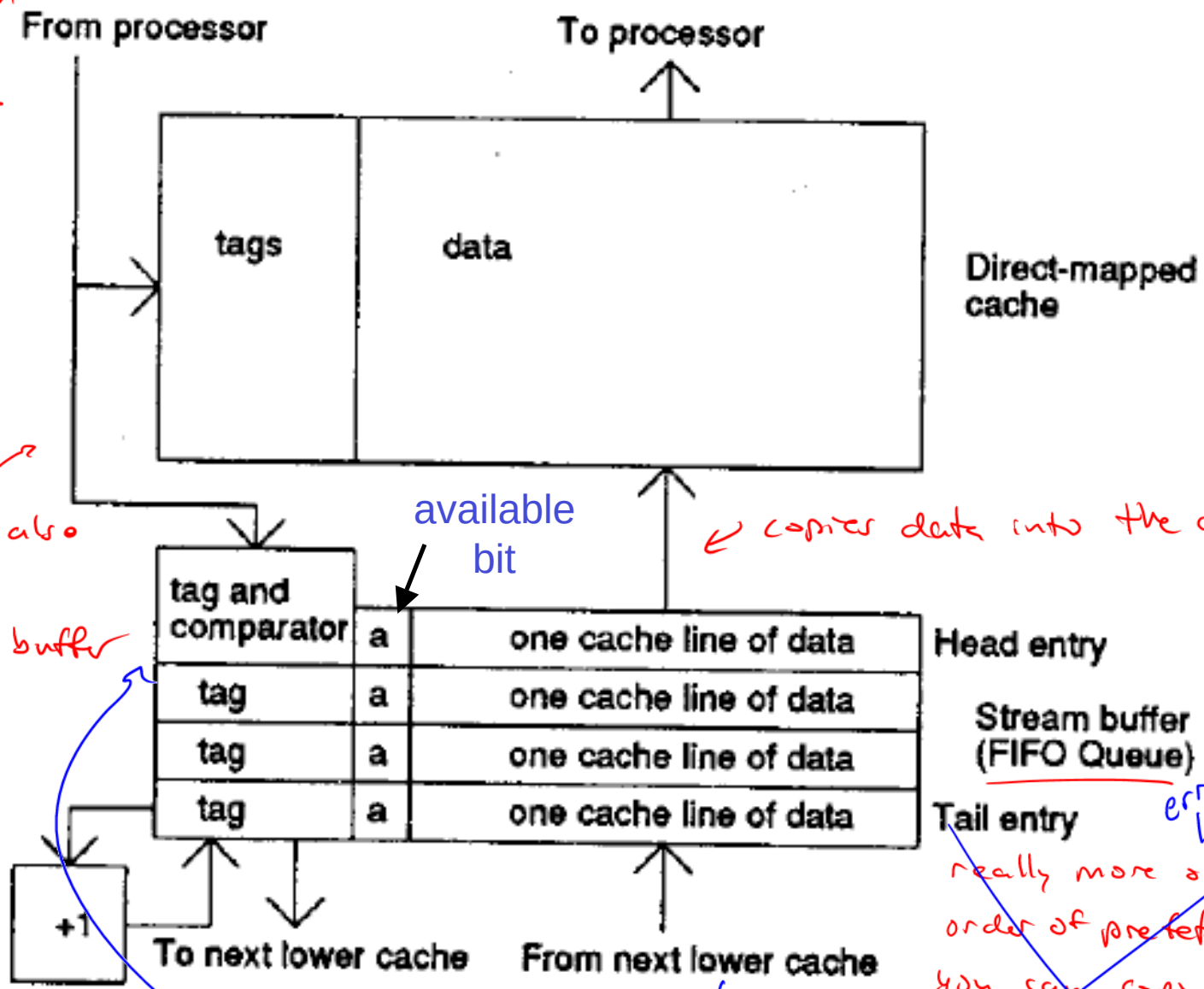
  → once the actual data is there, set the available bit to 1

- A cache miss that hits in the stream buffer causes the block to be loaded into the cache
  - Available bit must be 1 for there to be a hit!
  - Triggers further prefetching

    1. stream buffer → cache
    2. frees up space in the stream buffer
    3. pre fetch more stuff

Stream Buffer Organization

stream buffers are really targeted at compulsory "start-up" misses.



lookup also checks stream buffer

copies data into the cache

[Jouppi90]

strict FIFO!

if miss on head, flush queue regardless of stride / content

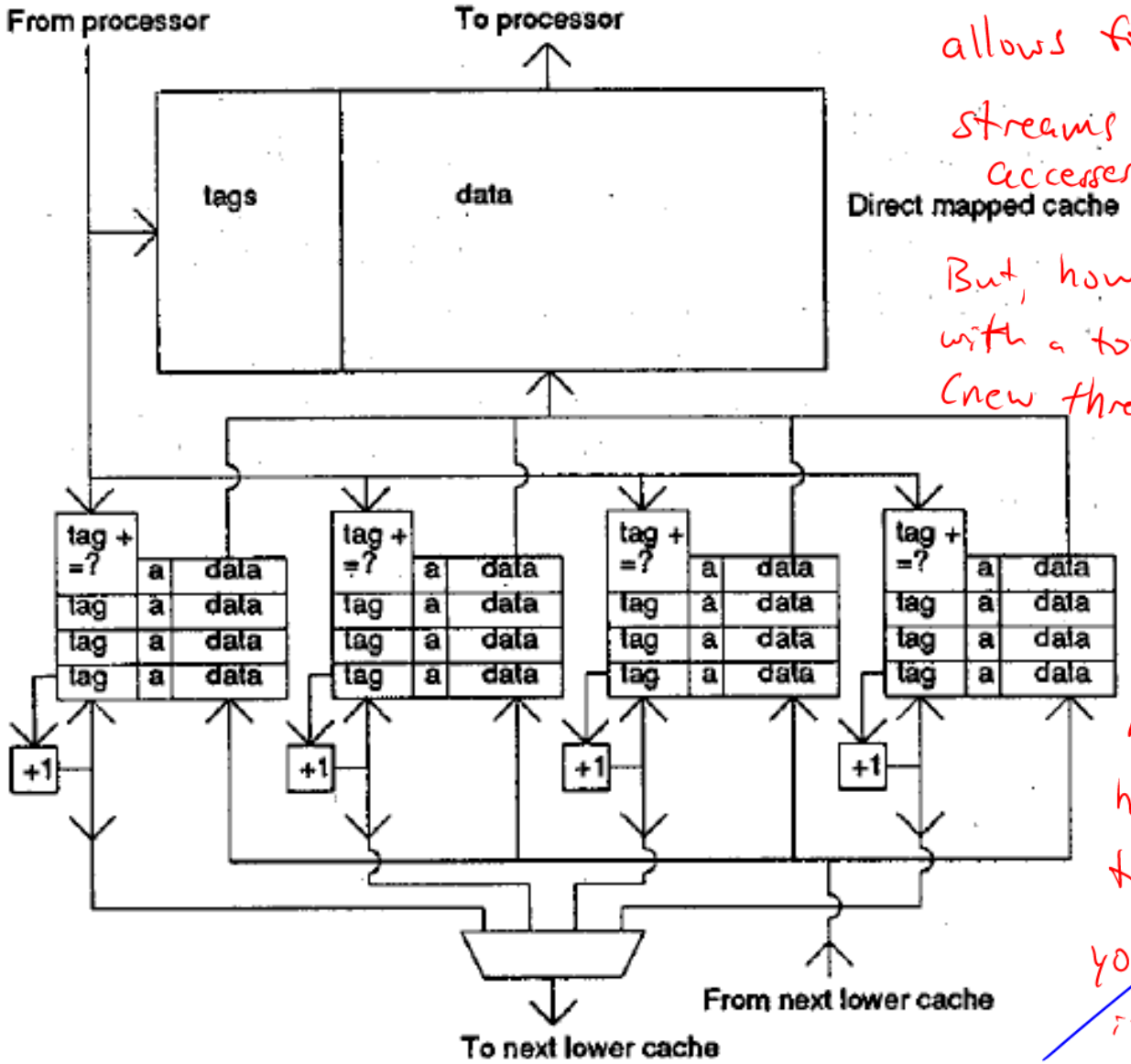
errata from lecture 6

~~really more of a order of prefetching you can copy any entry into the cache, but you always fetch in FIFO order~~

What if we have accesses that aren't contiguous? (i.e. matrix multiply)

n - way (n=4 in picture)

Multi-Way Stream Buffer



allows for n different streams in memory accesses.

But, how do you deal with a totally new stream (new thread or program, etc)

- potential solutions
  - some kind of aging counter

~~Assumption is you have enough streams to handle anything you can throw at it~~

see errata from lecture 6

[Jouppi90]

Non-Unit Stride Prefetching

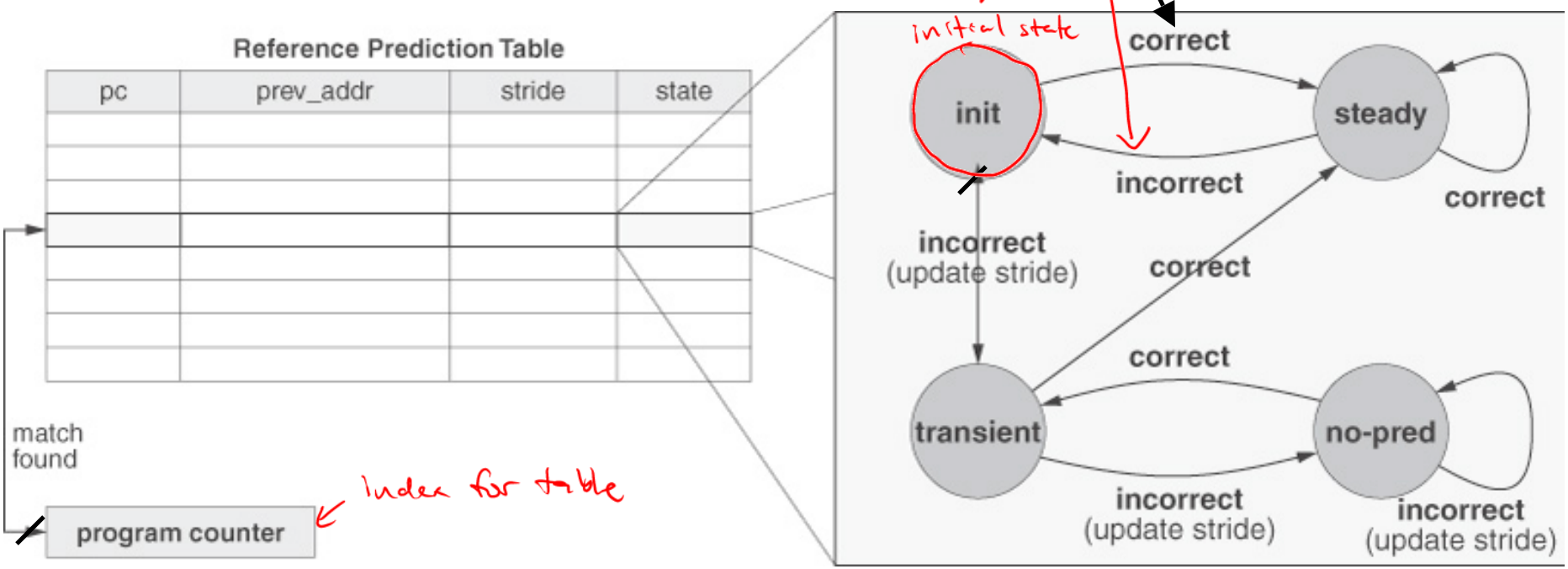
- *Stride predictor*: detects stride of accesses in order to predict the next block address
  - Example: Observe A, A+2 ⇒ predict A+4

- What do we need to track?
  - The last address generated by this load instruction
  - The last stride that we calculated based on the difference of the last two generated addresses
  - Whether we've seen enough of a pattern to make a prediction

Non-Unit Stride Prefetching

addr = prev_address + stride

≠



states:

Saturating state machine

init: new entry created or mis-prediction: stride = 0 @ initialization

transient: on the way to a prediction

steady: make prediction

no-pred: disable prefetching

Matrix Multiply Example

addr	instruction		comment	
500	lw	r4, 0(r2)	; load B[i,k]	stride 4 B
504	lw	r5, 0(r3)	; load C[k,j]	stride 400 B
508	mul	r6, r5, r4	; B[i,k] × C[k,j]	
512	lw	r7, 0(r1)	; load A[i,j]	stride 0
516	addu	r7, r7, r6	; +=	
520	sw	r7, 0(r1)	; store A[i,j]	stride 0
524	addu	r2, r2, 4	; ref B[i,k]	
528	addu	r3, r3, 400	; ref C[k,j]	
532	addu	r11, r11, 1	; increase k	
536	bne	r11, r13, 500	; loop	

one at a time (by word)

every 100th word

changes super slowly

```
int A[100,100],B[100,100],C[100,100]
for i = 1 to 100
    for j = 1 to 100
        for k = 1 to 100
            A[i,j] += B[i,k] × C[k,j]
```

[Baer91]

Matrix Multiply Example

B[0,0]

C[0,0]

A[0,0]

tag	prev_addr	stride	state
500	50,000	0	*init*
504	90,000	0	*init*
512	10,000	0	*init*

After iteration 1
(a)

B[0,1]

C[1,0]

A[0,0]

tag	prev_addr	stride	state
500	50,004	4	*transient*
504	90,400	400	*transient*
512	10,000	0	*steady*

After iteration 2
(b)

← start prefetching A
(don't need to, already in cache: stride = 0)

B[0,2]

C[2,0]

A[0,0]

tag	prev_addr	stride	state
500	50,008	4	*steady*
504	90,800	400	*steady*
512	10,000	0	*steady*

After iteration 3
(c)

start prefetching for B and C

yay!

[Baer91]

Next Time

Cache Content Management