Replacement policies
Prefetching

1 2   Compressed cache

L2 = 10 Misses

P 1000 → L1 → L2

100

10

10 × 1000 / 100

10.

0 1 1 0 0 0 1 1 0 1 0 1 0 0

LMR

10% - 100 = 10

A → 10% LMR

B → 10% GMR →

10% - 1000 → LMR.

GMR   Worse

LMR = 10/100 × 4-M

L2 → 100 Misses

→ 10% LMR.

10./100

128 bytes

Replacement algorithms

No choice in a direct mapped cache

In an associative cache, which way should be evicted when the set becomes full?

- Belady’s Optimal: L. A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-storage Computer. IBM Syst. J. 5, 2 (June 1966), 78–101.
- FIFO (first-in-first-out)
- LRU (least recently used), pseudo-LRU
- LFU (least frequently used)

Optimal Replacement Policy/ Belady's anamoly?

- [Belady, IBM Systems Journal, 1966]
- Evict block with longest reuse distance
  - i.e. next reference to block is farthest in future
  - Requires knowledge of the future!
- Can't build it, but can model it with trace
  - Process trace in reverse

- (X,A,B,C,D,X): LRU 4-way SA. How far in the future is X going to be accessed?

LRU

shift-reg.

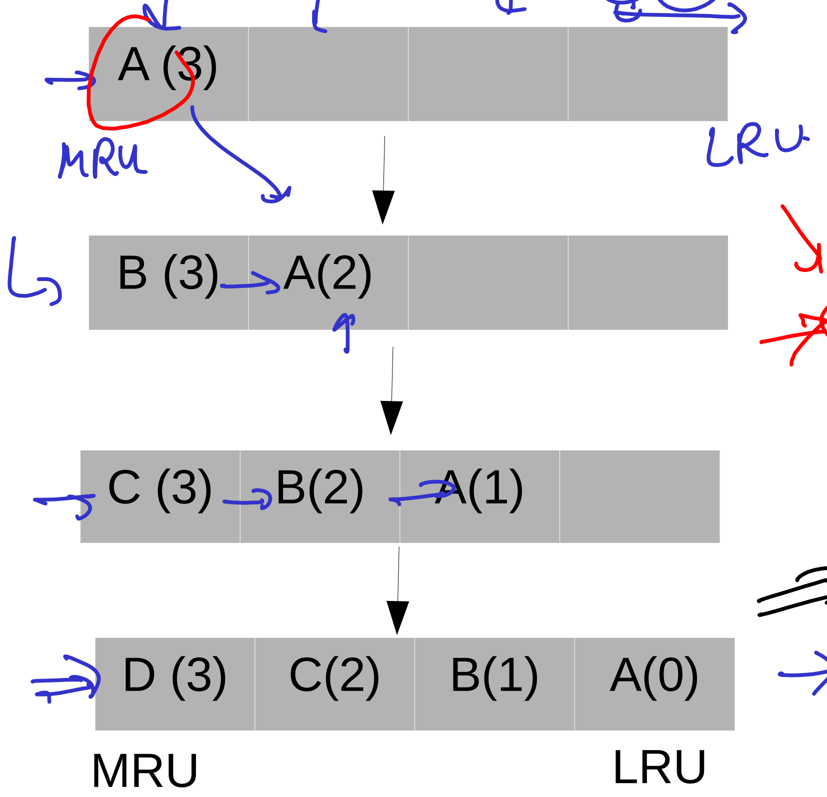
3- Most recent
0- Least recent

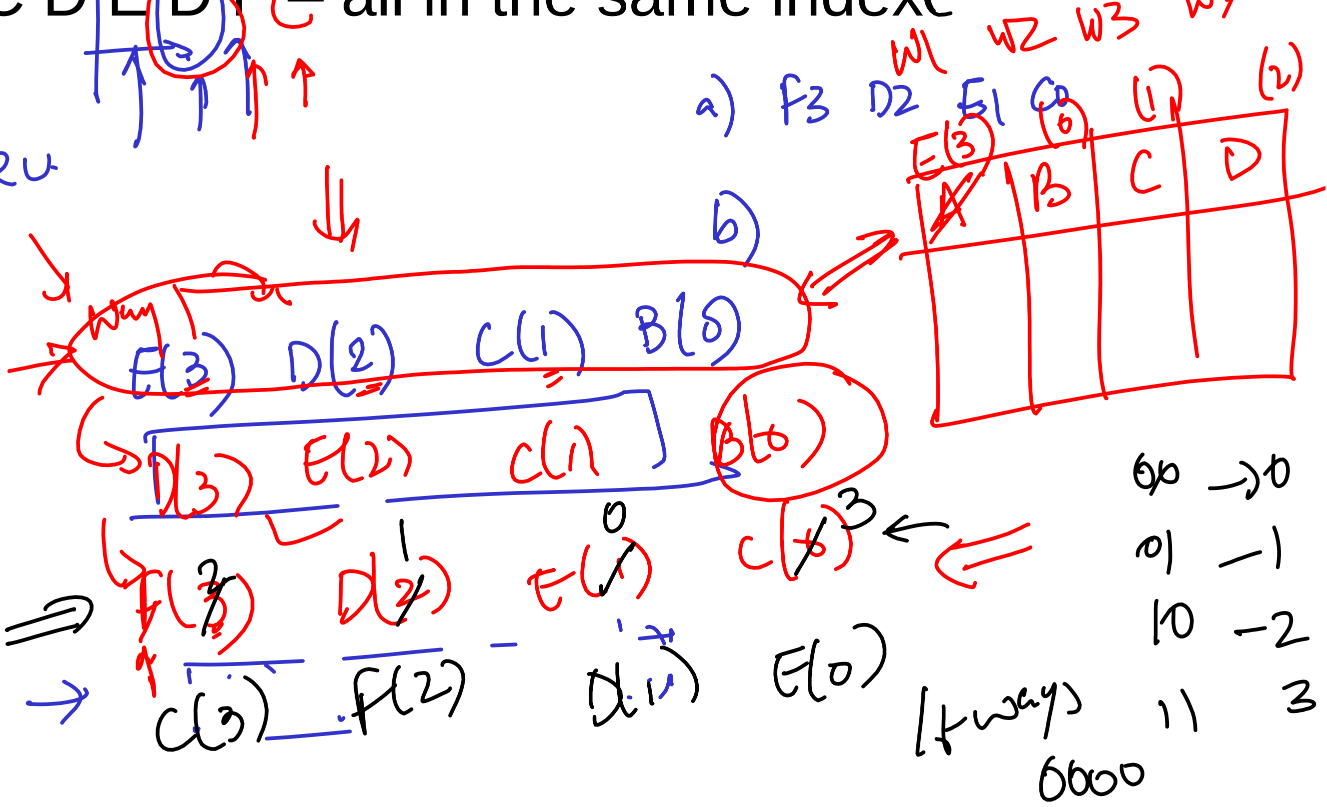
4 way SA cache

A B C D E D F C – all in the same index

W1(3) W2(2) W4(1)

LRU-Way



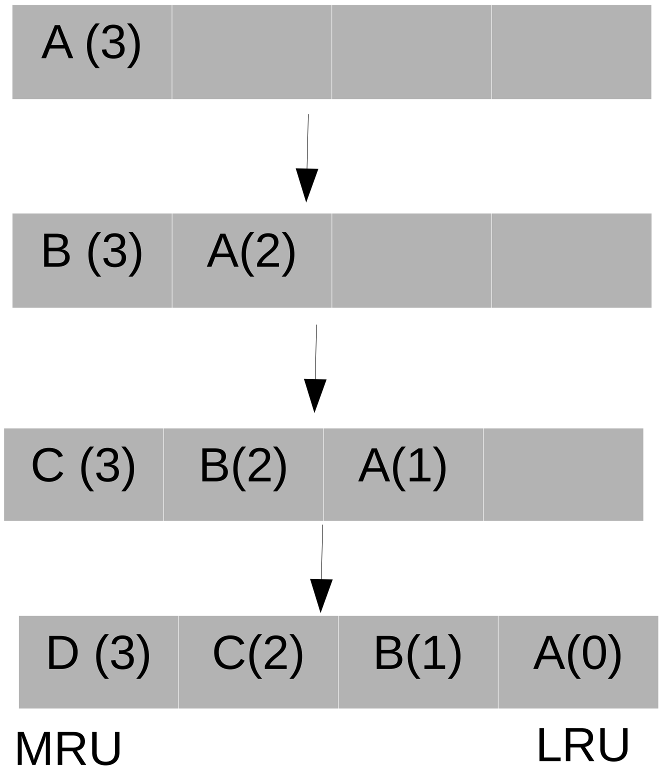


Maintain "age bits" or counters for cache-lines and replace based on the smallest number

LRU

3- Most recent
0- Least recent

A B C D E D F – all in the same index



A (3)			

B (3)	A(2)		

C (3)	B(2)	A(1)	

D (3)	C(2)	B(1)	A(0)

MRU LRU

MRU LRU

E (3)	D(2)	C(1)	B(0)

Replace LRU
Decrement
other counters

D (3)	E(2)	C(1)	B(0)

Update counters
Decrement if
more than the
current one

F (3)	D(2)	E(1)	C(1)

Update
counters

Maintain counters for each way.
Also, check all other counters if they are more than the current one

LRU with inclusive caches

To demonstrate back invalidation

A B A C A D A E A - accesses

A	B
0	1

Fully associative 2 blocks L1

LRU counter

1- MRU
0 - LRU

A	B	X	Y
2	3	0	1

Fully associative 4 blocks L1

LRU counter

3- MRU
0 - LRU

LFU

A B C D E C D B

Store the value of how many times it was accessed-frequency instead of when it was used.

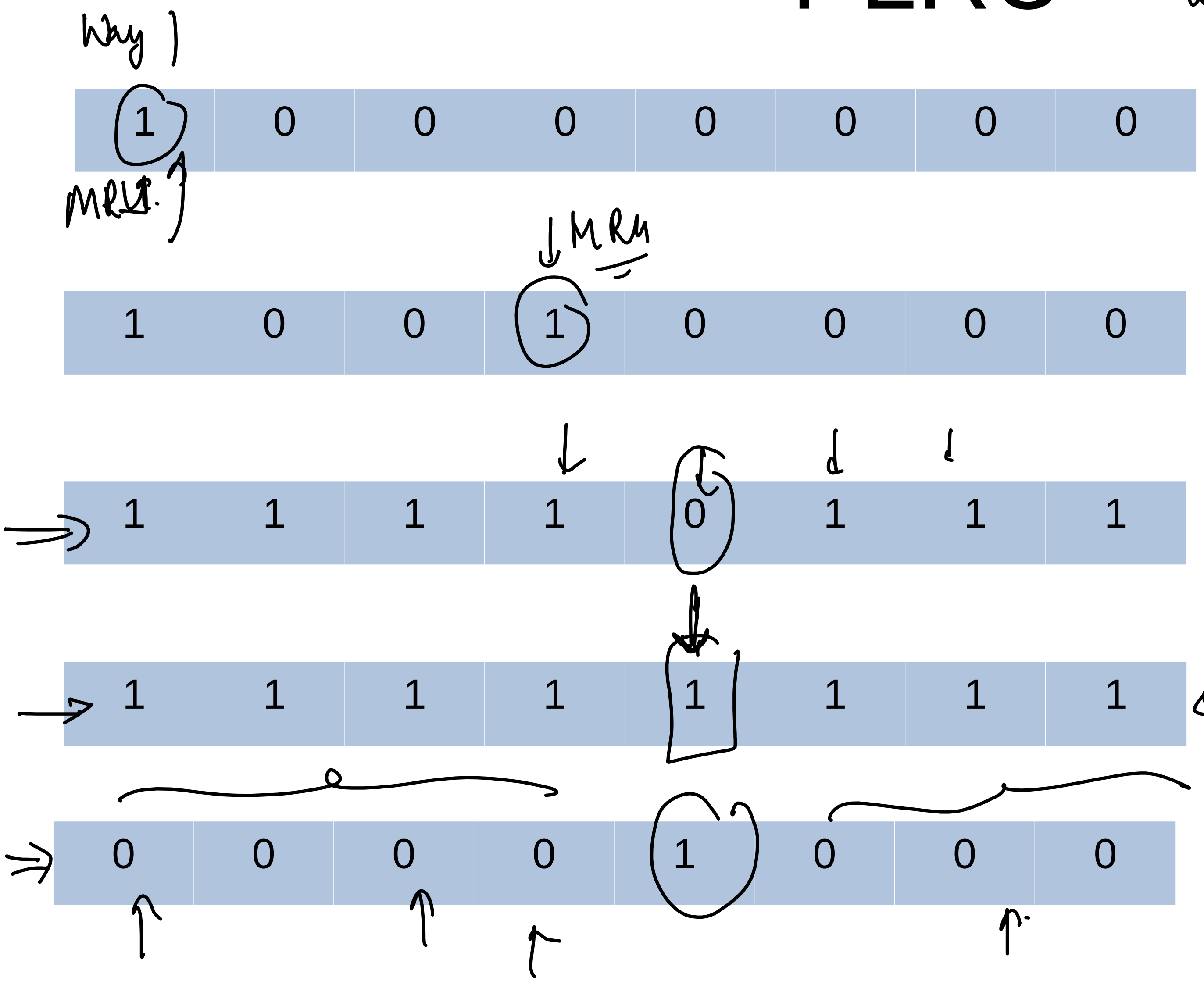
Can combine with LRU --> LFRU policy

Psuedo LRU

- LRU disadvantages -->
  - Counters for each block. 4 way: 2 bit counter + Keep track of other counters.
  - Update counters on each access
- Pseudo LRU: Single bit
  - 1 – when accessed
  - 0 – is the least recently accessed
  - Replace the block which has a 0 bit (randomly if more than one block which has ‘0’)

PLRU

64 ways + 6-bit



1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
1	1	1	1	0	1	1	1
1	1	1	1	1	1	1	1
0	0	0	0	1	0	0	0

Set a block bit to ‘1’ if accessed.

Random

On a hit, no need to check bits of other blocks.
Just set the block to ‘1’

When the last one is set to ‘1’, zero out the others

Replace a block which has ‘0’ bit, and set the bit to 1. This is done randomly, so not true LRU

4-way superscalar

IF ID EX M W
IF ID
IF ID
IF ID

IF ID
IF ID
IF
IF

LW a(i) LW
LW b(i) LW
MUL MUL
add add
SW SW

4
IF ID

3

1 2

4
Add i+4
Beq

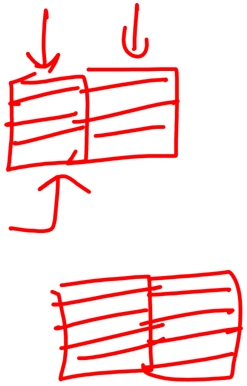
A[0] B[0]
A[1]
A(2)
A(31) B(31)

L1

L2

L3

MM.



Cache Prefetching

- Steven P. Vanderwiel, “Data Prefetch Mechanisms” ACM 2000
- H&P – Chapter “Memory Hierarchy design”

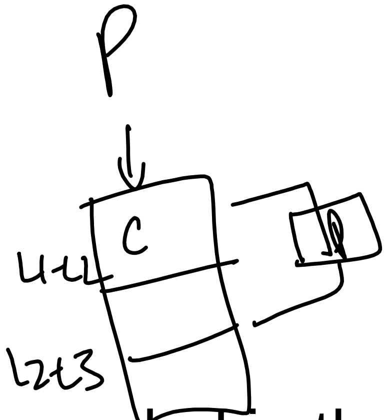
Motivation

- Cache: “on demand” memory fetch policy
  - processor requests a word --> cache miss --> fetch
  - Store only previously accessed data
  - Larger block size --> fetches consecutive words, but on-demand
    - Disadvantage: Evict all blocks (evict useful data)
- Anticipates cache misses and issues a fetch to the memory system--> placed in a prefetch buffer
- Proceeds in parallel to processor computation
- Significantly improve overall program execution

Introduction

- Speculate: Fetch instructions or data from memory to cache before they are needed
- Data or instruction prefetching
  - Easier to guess instructions
- How?
  - Software based: Programmer/compiler inserts "prefetch" instructions in the program
  - Hardware based: In processor: watches the stream of instructions or data

Introduction



- What to fetch:
  - Involves predicting which address will be needed in the future
  - Misprediction: Is ok. Prefetched data will not be used
  - Predict based on past patterns
  - Prefetching algorithm
- PreFetch data up in the memory hierarchy before they are actually needed by the processor

Metrics

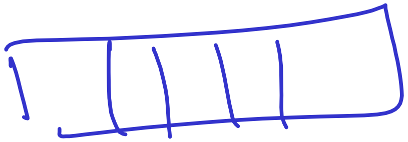
Prefetches

- Accuracy = No of useful prefetches / Total prefetches
- Timeliness = When to prefetch?
  - Too early
    - Might replace useful data
    - Might get replaced by the time it is used (Stall)
  - Too late
    - Processor has to wait (Stall)

p

Hide → Mem access latency

LRU.



8

Timeliness

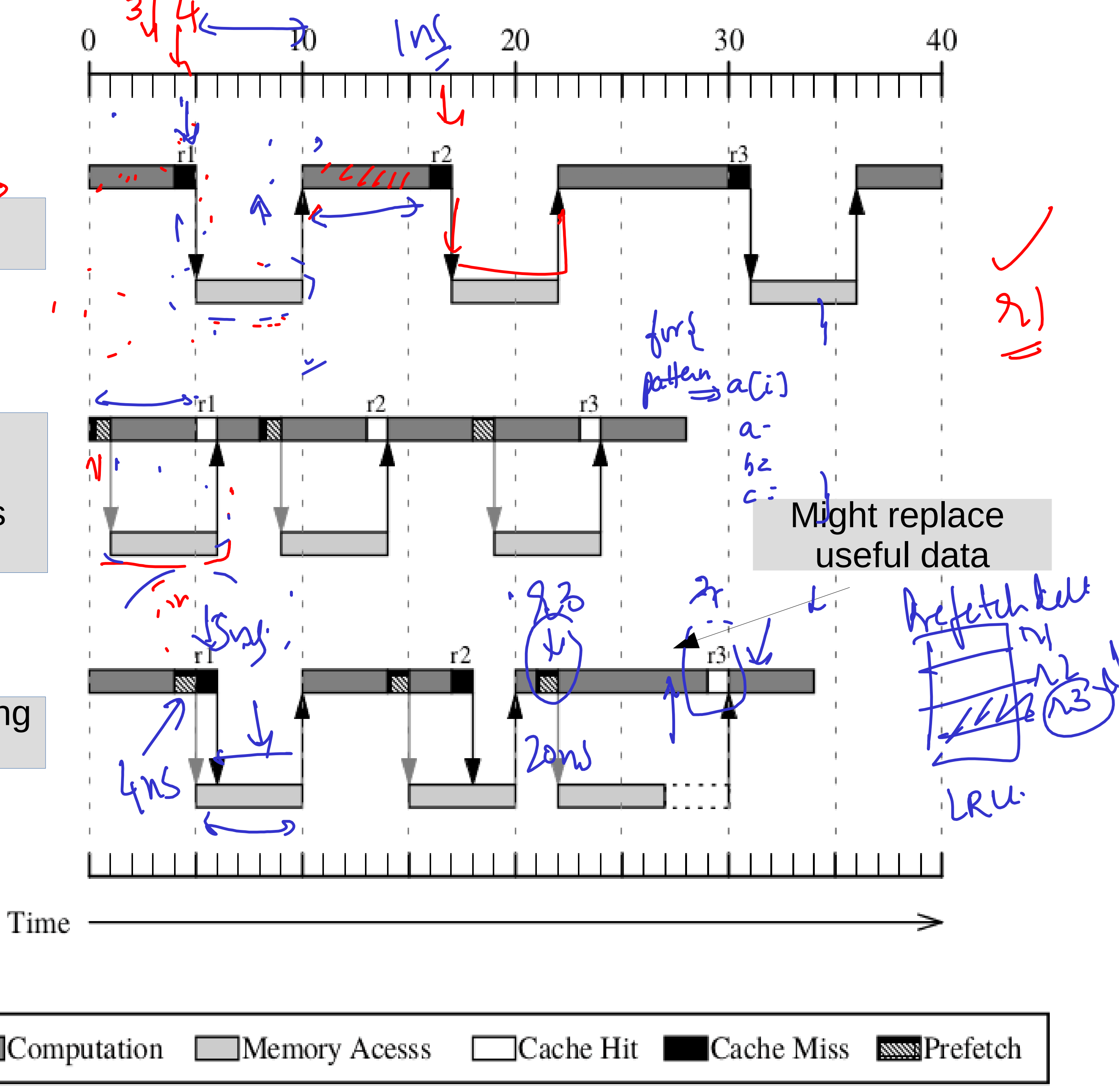
no prefetching

P → Computation Prefd

perfect prefetching-
No misses
Fetch “memory cycles
Latency- earlier”

late

degraded prefetching
Prefetch late



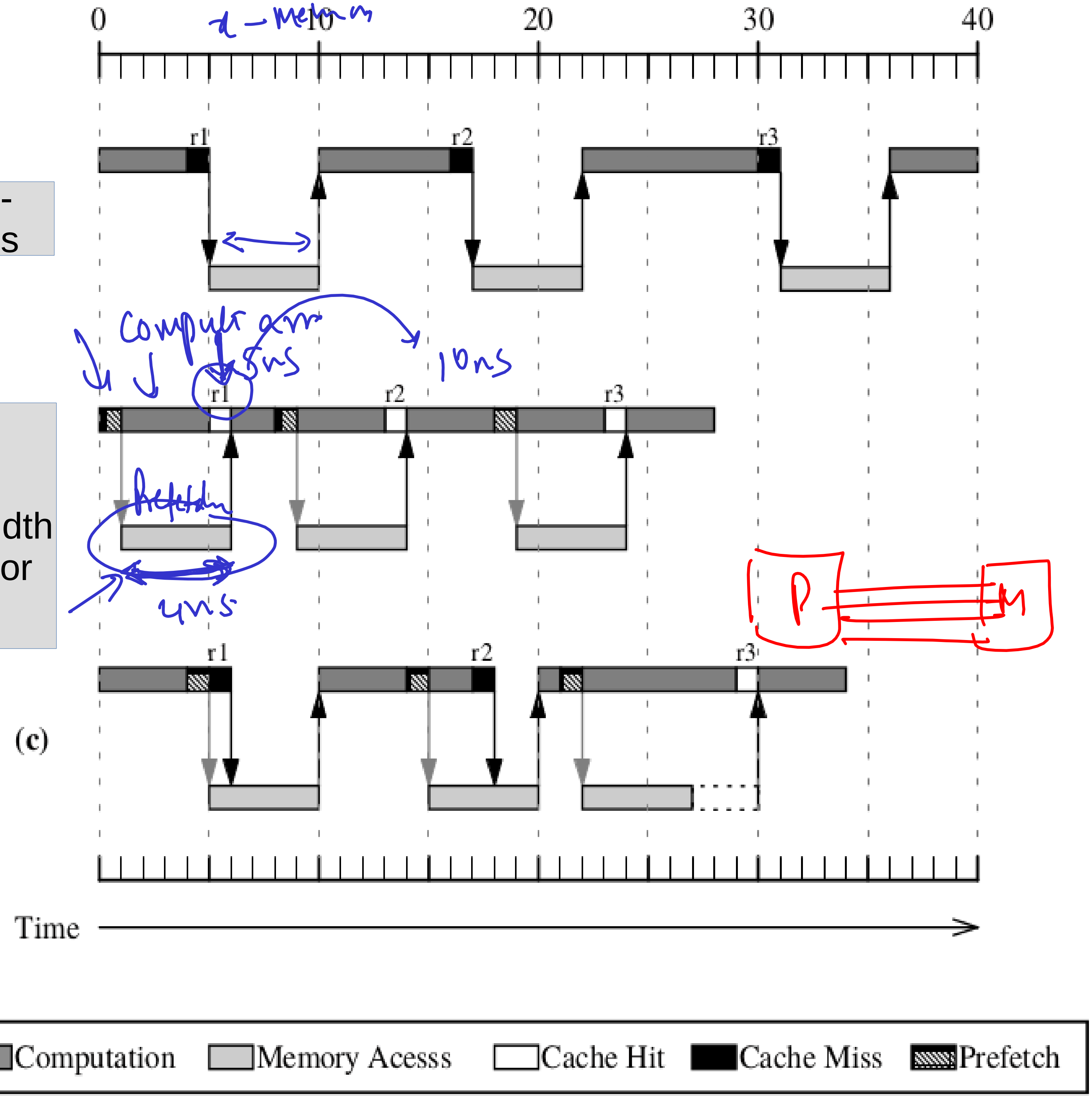
Might replace
useful data

Steven P. Vanderwiel, “Data Prefetch Mechanisms” ACM 2000

Memory accesses

31 units of time--
3 mem accesses

19 units of time--
3 mem accesses
Higher Memory bandwidth
Work done by processor
is more

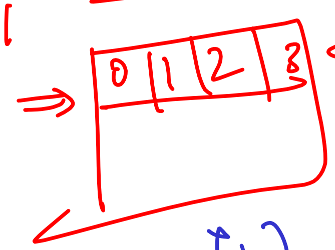


Steven P. Vanderwiel, “Data Prefetch Mechanisms” ACM 2000

Software prefetching

Assume a Cache with ~~4~~-word blocks

4 Bytes



=> a[4]

a[8]

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

i+2

a[i+1] * b[i+1]

no prefetching
Will miss every ~~4~~ 4th iteration

0 1 2 3

1 2 3 4

a[4]   prefetch buffe

```
for (i = 0; i < N; i++){
    fetch( &a[i+1]);
    fetch( &b[i+1]);
    ip = ip + a[i]*b[i];
}
```

One. =

a[0]   a[1]

simple prefetching
Prefetch every iteration --> fetches 4 blocks

0 1 2 3

4 5 6 7

```
for (i = 0; i < N; i+=4){
    fetch( &a[i+4]);
    fetch( &b[i+4]);
    ip = ip + a[i]*b[i];
    ip = ip + a[i+1]*b[i+1];
    ip = ip + a[i+2]*b[i+2];
    ip = ip + a[i+3]*b[i+3];
}
```

loop unrolling

Parallel

a[4] = i+4

fetch(&a[i+5]
i+6
i+7
i+8

prefetching with loop unrolling
Unroll y times, where y = number of words in each block

loop → LW a(i)
→ LW b(i)
→ MAC SW
LW a(i+1)

Cache misses will occur during the first iteration of the loop
Unnecessary fetches during the last iteration
This code does not consider memory latency

{ LW b(i+1)
MAC
SW -
LW i+2
LW i+3
→ Add i+4
Beg loop

Software prefetching for loops

```
for (int i=0; i<1024; i++) {
    array1[i] = 2 * array1[i];
}
```

Works well with loops and regular access patterns

i=7

k=7

```
for (int i=0; i<1024; i++) {
    prefetch (array1 [i + k]);
    array1[i] = 2 * array1[i];
}
```

i=0 →
1

0
1
2
:
6
7

7
8
7

0+1

0+k

7

Prefetch k elements ahead

K= Prefetch distance ←

What is k?

If k=7, Compulsory misses: 0 to 6 will be misses

- Prefetch instructions added either by the programmer or compiler
- Prefetch for the next iteration
- Prefetch directives improve performance

Next-line
i+1

Strided prefetching

Offset prefetch.

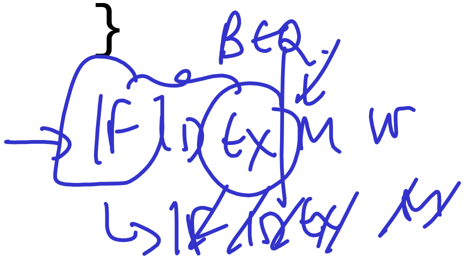
Software prefetching for loops

```
for (int i=0; i<1024; i++) {
    array1[i] = 2 * array1[i];
}
```

i+k

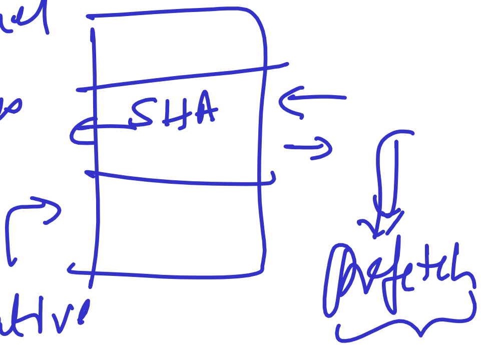
Patches {Meltdown, Spectre}

Cache i+nk



Works well with loops and regular access patterns

Side channel attacks



```
for (int i=0; i<1024; i++) {
    prefetch (array1 [i + k]);
    array1[i] = 2 * array1[i];
}
```

Speculative

Memory bound

- Currently i=0--> k=1, Prefetched data might not be available by the next iteration. (Too late)

- i=0, k=20, Prefetched data might come in too early and replace useful data (Too early)

Not easy to predict k at compile time

- Can combine loop unrolling with software prefetching

Software prefetching for loops

Prefetch how many iterations ahead?

- L/S iterations ahead

hide.

L = average memory latency in processor cycles

S= shortest execution time of 1 iteration

$$\frac{5 \text{ cycle.}}{1}$$

L = 5 cycles of memory latency

S= 1 cycle

Prefetch how many iterations ahead?

- 5 iterations ahead

Limitations

- + − Works for Regular and predictable array accesses
- − Code expansion, more instructions

  exception
  page fault

- − Statically done by compiler, does not exploit runtime information
- − Programmer has to do this manually

Hardware prefetching

a+1 ←

- Sequential prefetching
- Strided (distance) prefetching

→ lw a[0] ←

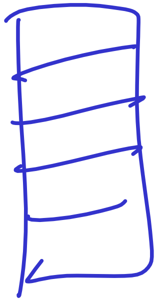
a+k ←

→ Best offset prefetcher (BOP)

a+10

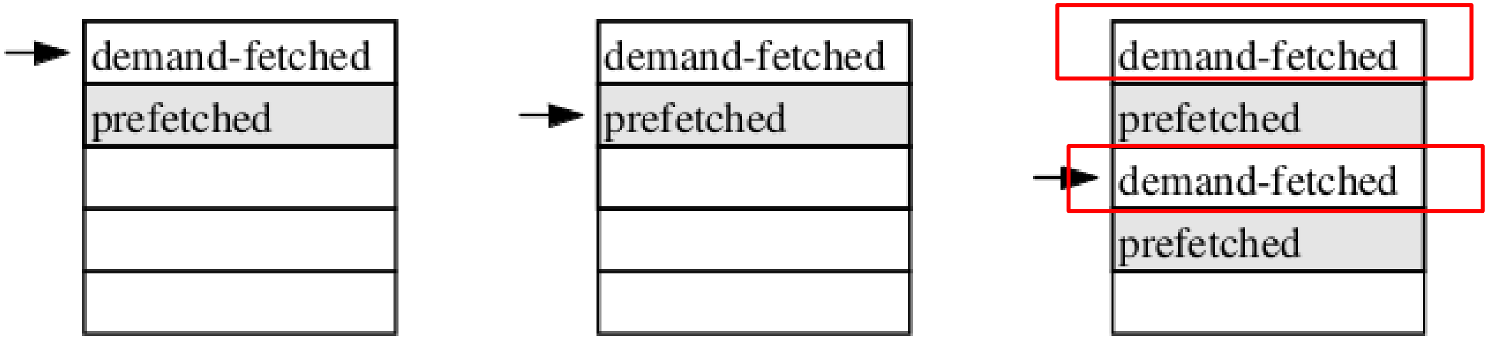
a+5

a[i+1]



Sequential prefetching

- Prefetch on miss
  - Prefetch next block y+1 if y is a miss
  - If y+1 is already in cache, do not prefetch

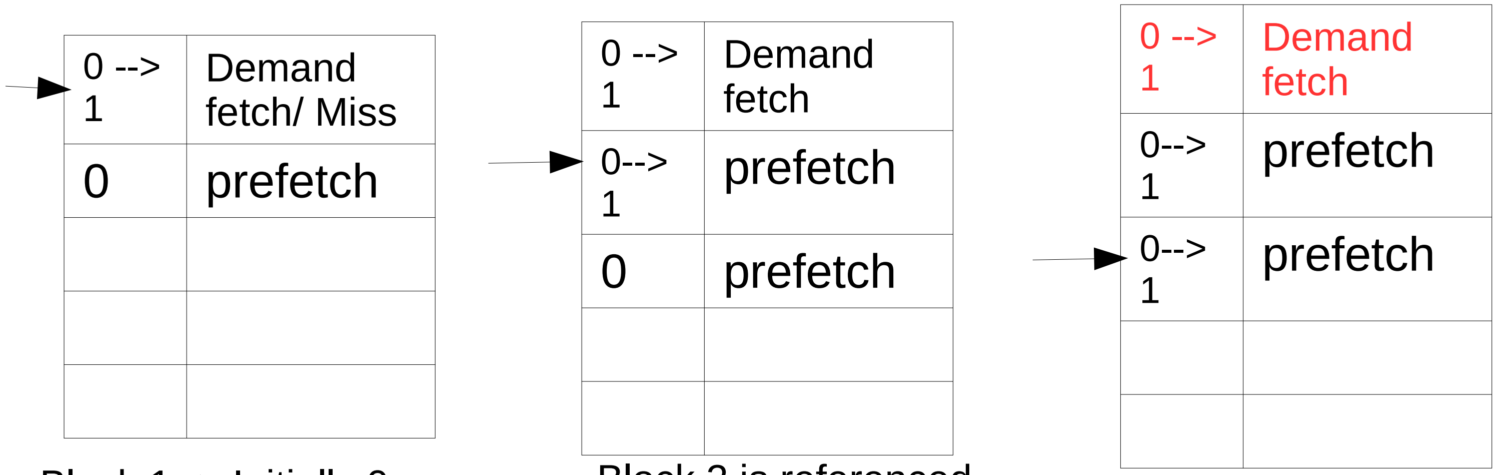


Demand fetch means miss
No prefetch generated here since
it was a hit

Overall 2 misses

Tagged prefetching

- A tag bit with each cache line
- 0:
  - Initially
  - Reset to 0 on replacement
  - Prefetch
- 1:
  - When the line is referenced
  - Brought into cache on demand (after a miss)
- **Prefetch is initiated when tag changes from 0 --> 1, that is, if the line is referenced (indicated by arrow)**
  - <u>When a data is prefetched and accessed, more confidence in prediction</u>



0 --> 1	Demand fetch/ Miss
0	prefetch

0 --> 1	Demand fetch
0--> 1	prefetch
0	prefetch

0 --> 1	Demand fetch
0--> 1	prefetch
0--> 1	prefetch

Block 1--> Initially 0, changed to 1
Initiate prefetch for next block

Block 2 is referenced and is a hit. 0--> 1
Initiate prefetch for next block

Overall 1 miss

Strided prefetching

- Strided (distance) prefetching: Consecutive blocks that are fetched are “y” addresses apart
  - Block b, b+ y, b+ 2y?
- Check 2 consecutive loads, and the distance between their memory accesses. Set this as the stride
  - Predict their access pattern

Where to place the prefetched data - is a design choice
H/W based stream buffers

- Placing in cache is easiest --> Pollutes the cache. Replaces useful data
- Buffers: k subsequent addresses are fetched into a buffer of depth k:
  - Eg- k=4--> Miss on A fetches A+1, till A+4 – **Sequential/ Next-line prefetching**
- Multiple streams for multiple misses

↓ Power
↓ Space

1) Accurate prefetcher

2) Aggressive

a+1
10

b+k
b+k+10

Accurate

L1-L2

L2-L3

~~A~~ →LRU
B
C
D

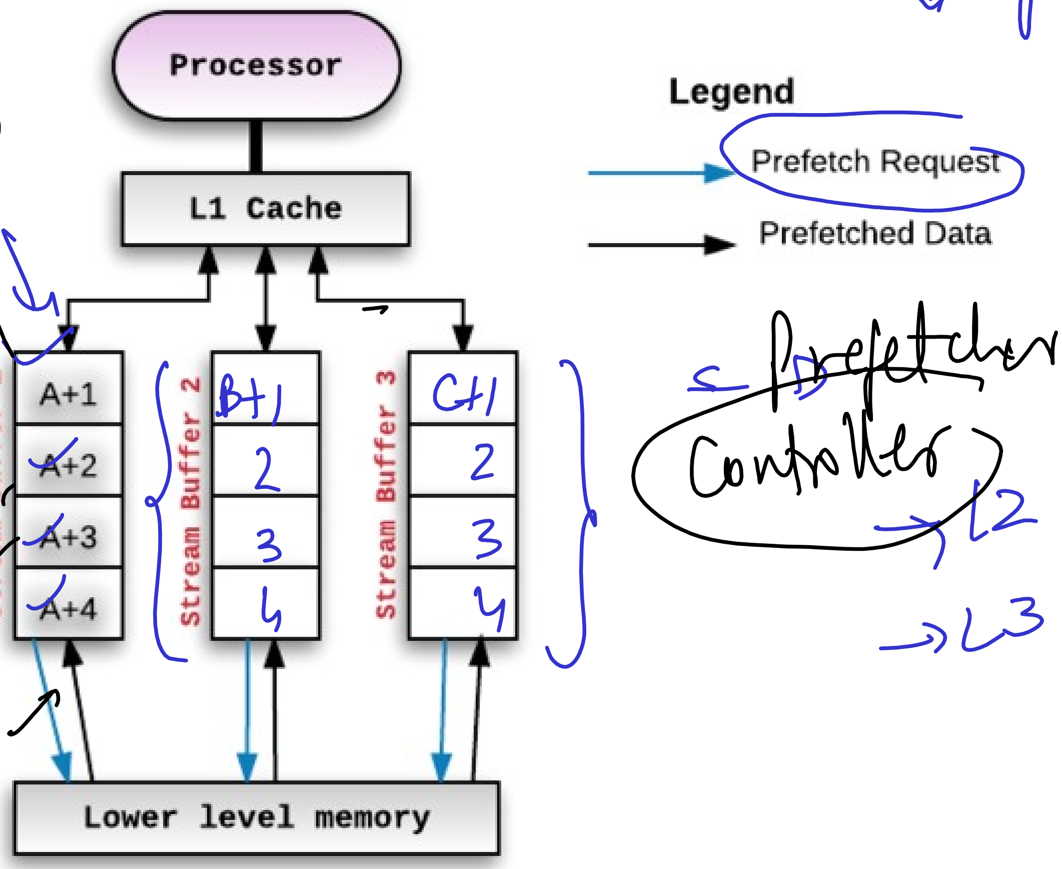
4 bytes

a[ ]

0 1 2 3

4 5 6 7

↓
a[0]

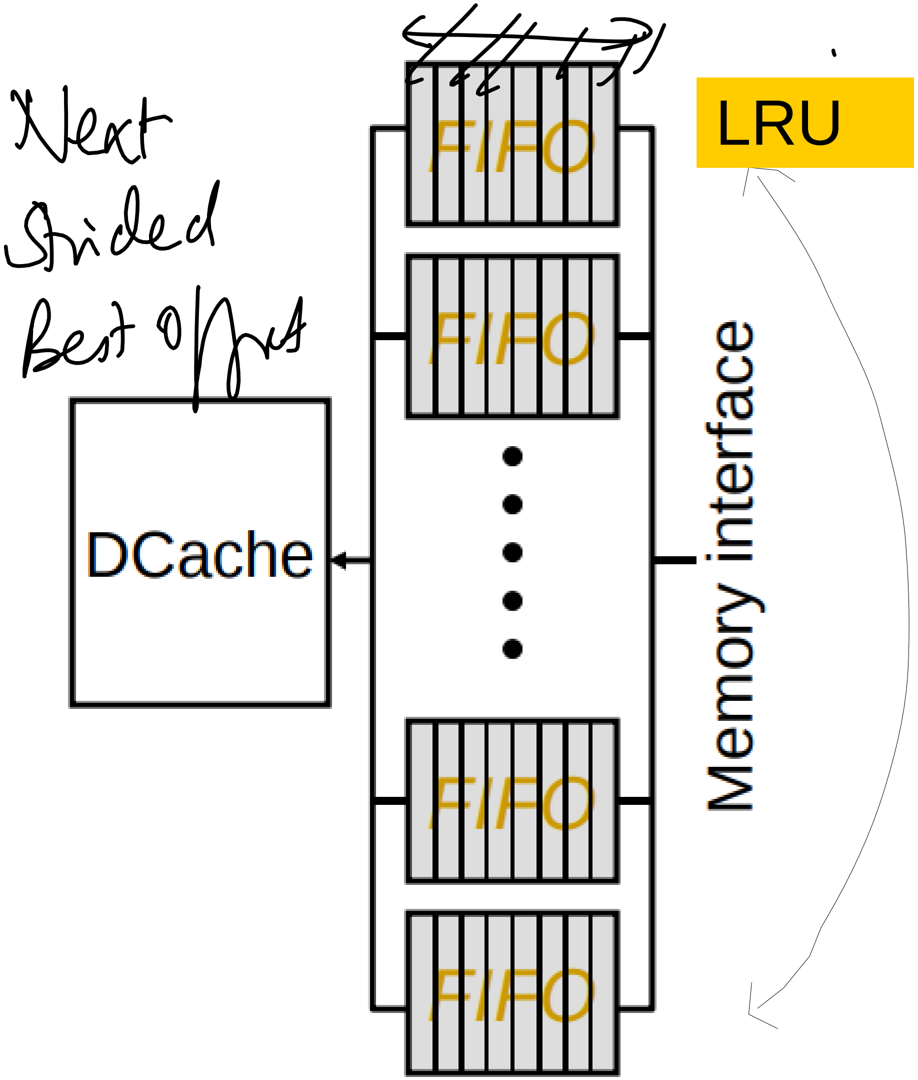


Prefetcher
Controller
→L2
→L3

Wiki

Multiple Stream buffers for data

Next
Strided
Best Offset

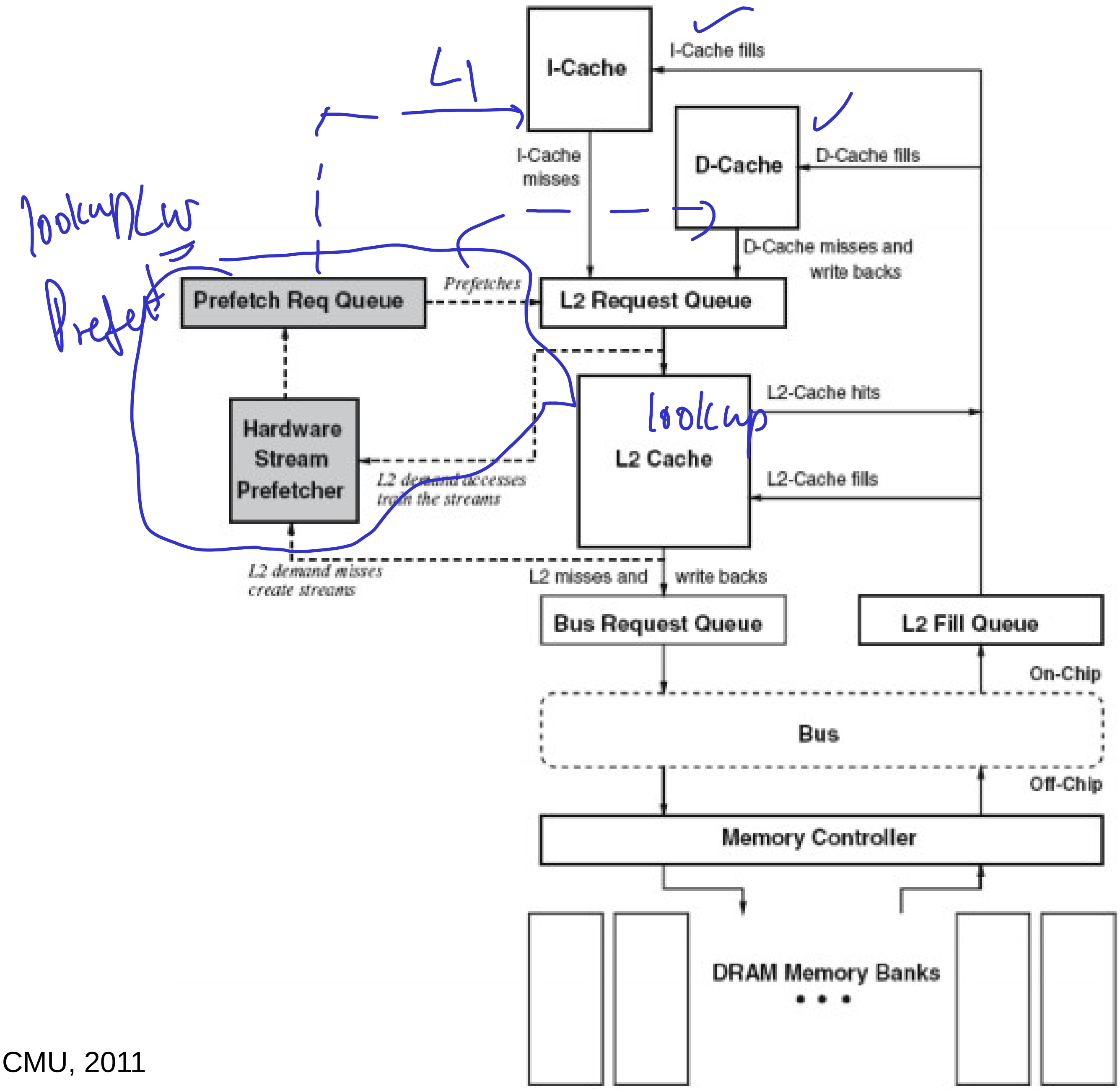


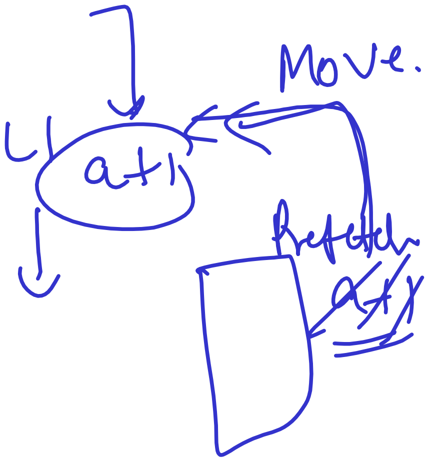
- Each stream buffer holds one stream of sequentially prefetched cache lines
- On a cache/load miss, check the head of all stream buffers for an address match
  - Hit, pop the entry from FIFO, update the cache with data. Fetch the next in line data
  - Miss-- allocate a new stream buffer to the new miss address
  - If all streams are full, replace a stream buffer following LRU policy
- L2 and stream buffer can be checked parallelly

Buffers

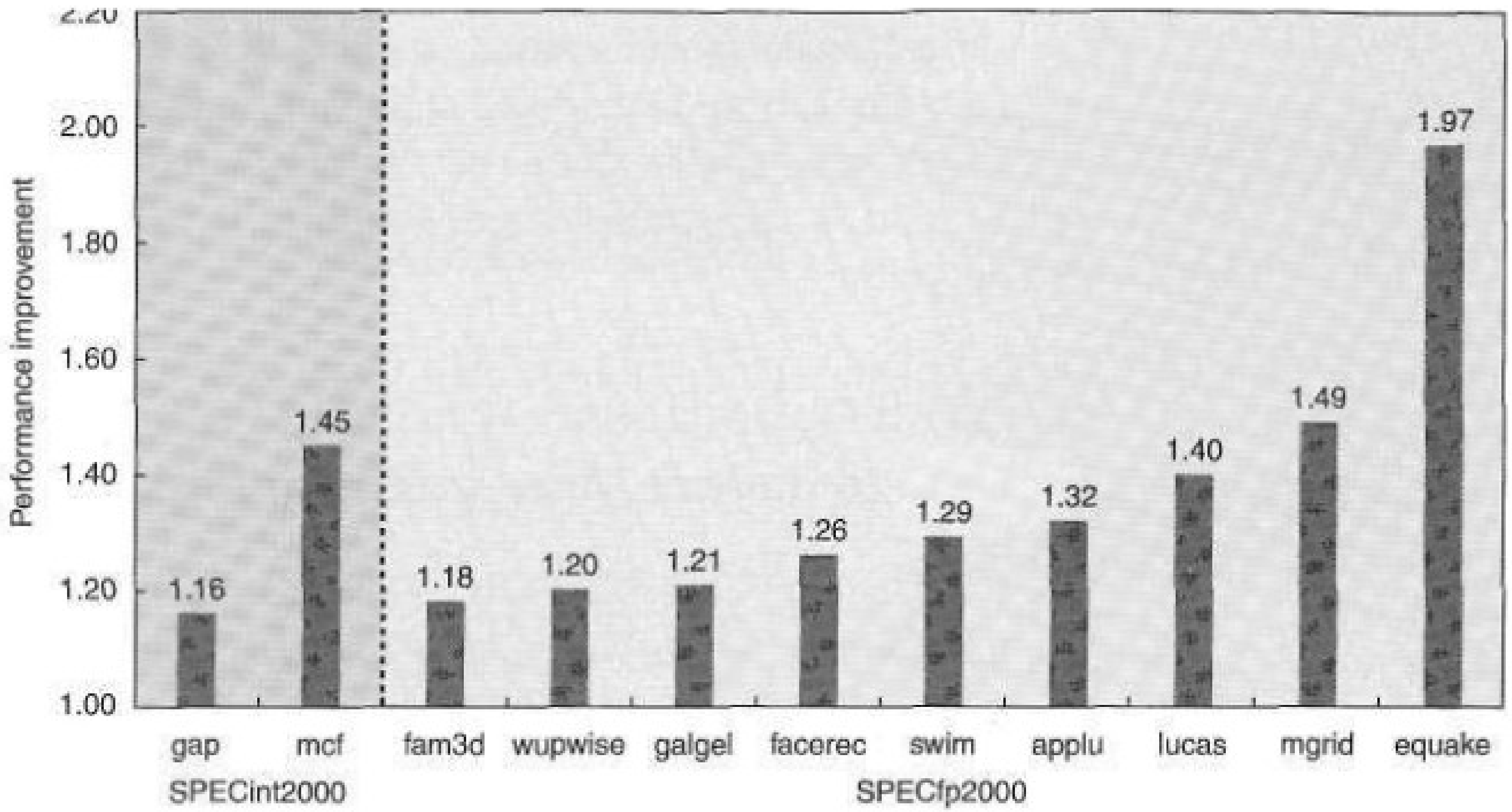
- Where to place the buffer: btn L1 and L2, memory to L2?
- When to access the buffer (parallel vs. serial with cache)
- When to move the data from the prefetch buffer to cache
- Size the prefetch buffer

Overall system





Performance



Speedup due to hardware prefetching on Intel Pentium 4 with hardware prefetching turned on

Acknowledgements

- TU Berlin, Software and Hardware Prefetching
- UCB- CS 152 https://inst.eecs.berkeley.edu/~cs152/sp16/lectures/L06-Memory.pdf