Speculation in memory

Hiding memory latency from the program

When is data brought into the cache?

On-demand

- A cache block is only filled when a miss for that block occurs
- In an L1 cache, misses originate from load and store instructions

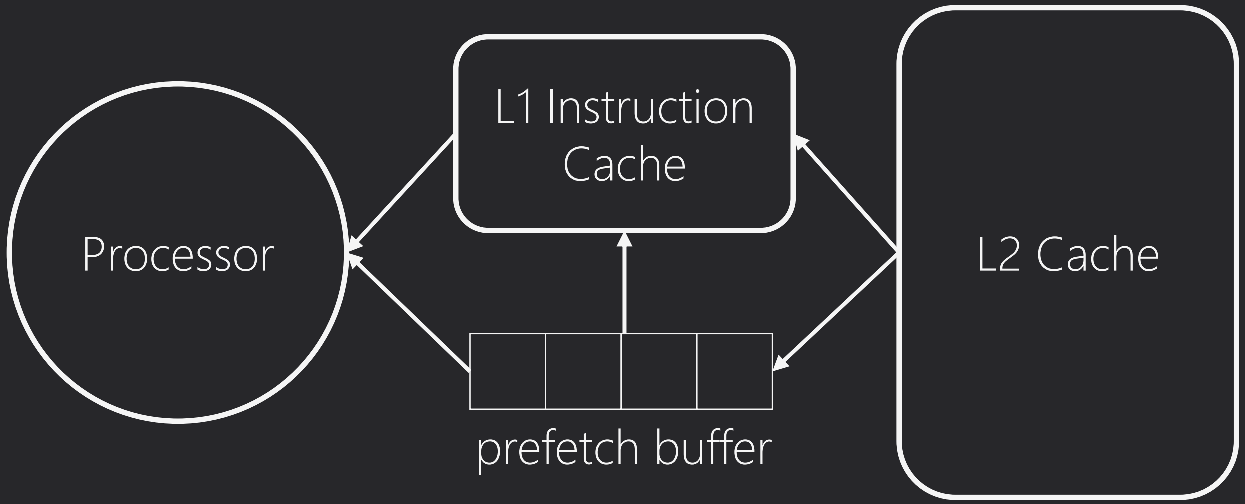
Prefetch

- A cache block may be filled due to speculation
  - i.e., predict that this cache block is needed in the “near future”

What is prefetching?

- A predictive technique that brings data into a cache before it is needed
- Prefetching can be done…
  - For instructions or data (or both)
  - Between any two levels of the memory hierarchy

- Goal: Increase the hit ratio of the cache



What if the prefetcher mispredicted?

- Mispredictions do not impact program correctness

- Mispredictions may result in worse performance
  - e.g., the prefetch evicted a useful cache block

- Can prefetchers adapt to mispredictions?
  - Possible, but feedback mechanism is difficult and lengthy

How are prefetchers assessed?

- $Accuracy = \dfrac{useful\ prefetches}{number\ of\ prefetches}$

- $Coverage = \dfrac{useful\ prefetches}{number\ of\ misses\ (without\ prefetcher)}$

- Timeliness
  - Too early: evicted useful data from the cache
  - Too late: a demand access had to wait (though the prefetch reduces the latency)
  - Too early (worst-case): two evictions (one for useful data, one for prefetched data)

What are the disadvantages of prefetching?

- Prefetchers compete for hardware used by regular memory operations

- e.g., should not prefetch lines that are already in the cache
  - Contention: cache tags

- Cache pollution
  - e.g., evictions and "Too early (worst-case)" behaviour

- Bandwidth pollution
  - e.g., need to send requests to next level of the hierarchy (via a bus). Priority to the bus must be given to load and store instructions

What is a sequential prefetcher?

- Prefetch an adjacent cache block to *i*
  - Forward: *i + 1*
  - Backward: *i - 1*
- When to prefetch?
  - Always prefetch
  - Prefetch on miss
  - Other?
- What is the prefetch degree?
  - The number of blocks to prefetch *N*

🟥 Fetch *i*

🟩 Prefetch *i + 1*

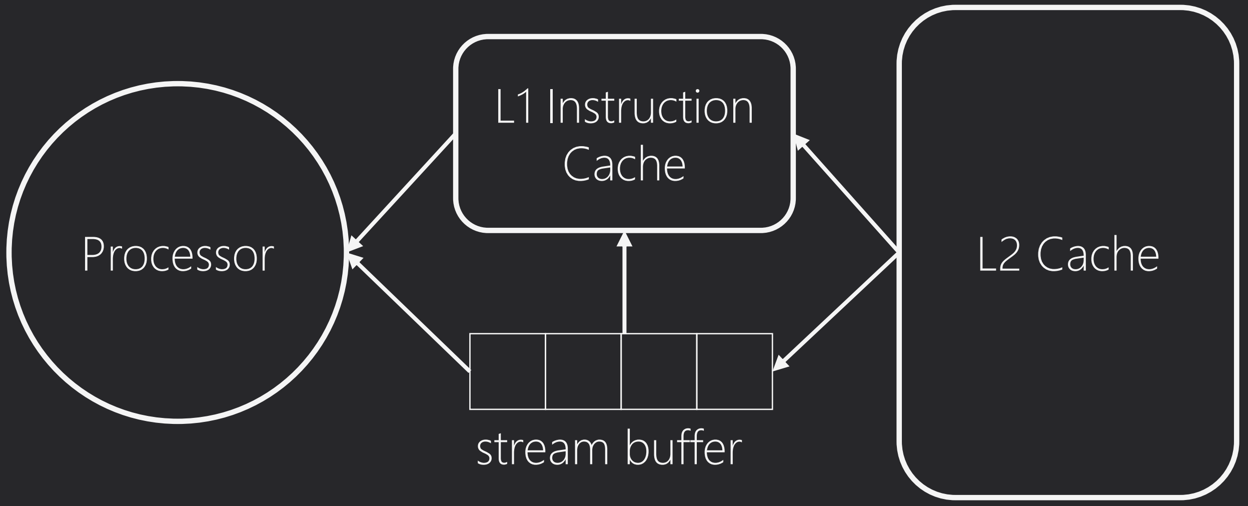
Address	*Instruction*
🟥 500	`LW r4, 0(r2)`
🟥 504	`LW r5, 0(r3)`
🟥 508	`MUL r6, r5, r4`
🟥 512	`LW r7, 0(r1)`
🟩 516	`ADD r7, r7, r6`
🟩 520	`SW r7, 0(r1)`
🟩 524	`ADD r2, r2, 4`
🟩 528	`ADD r3, r3, 400`
532	`ADD r11, r11, 1`
536	`BNE r11, r13, 500`

Example: Alpha AXP 21064

- Add a *stream buffer* to hold prefetched data

- On access,
  - Check cache *and* stream buffer
  - If found in stream buffer, move to cache

- On miss,
  - Fetch missed block *i* into cache
  - Prefetch block *i+1* into stream buffer iff it is not already in the cache



Analysis of sequential prefetching

Advantages

- Adapts to different programs (and phases of programs)
  - Hence, different from just increasing block size
- Stream buffer avoids cache pollution
  - Only move data to cache when it is accessed
- Great for instruction caches (sequential access pattern)

Disadvantages

- Timeliness
- Memory accesses patterns are not always sequential
- Stream buffers have some overhead
  - For instruction cache, one stream buffer is usually sufficient
- Poor for data caches

Effective hardware prefetching

Chen, Tien-Fu, and Jean-Loup Baer. "Effective hardware-based data prefetching for high-performance processors." IEEE transactions on computers 44.5 (1995): 609-623.

How do we classify memory access patterns?

```
int A[100][100];
int B[100][100];
int 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]
```

- Scalar
  - e.g., reference to `i`
- Zero stride
  - e.g., `A[i][j]` is the same in iterations of the inner-most loop
- Constant stride
  - e.g., `B[i][k]` in the inner loop is accessing `B[i][0], B[i][1], …, B[i][100]`
- Irregular
  - (not shown)
  - e.g., for linked lists

Example: matrix multiply's inner loop

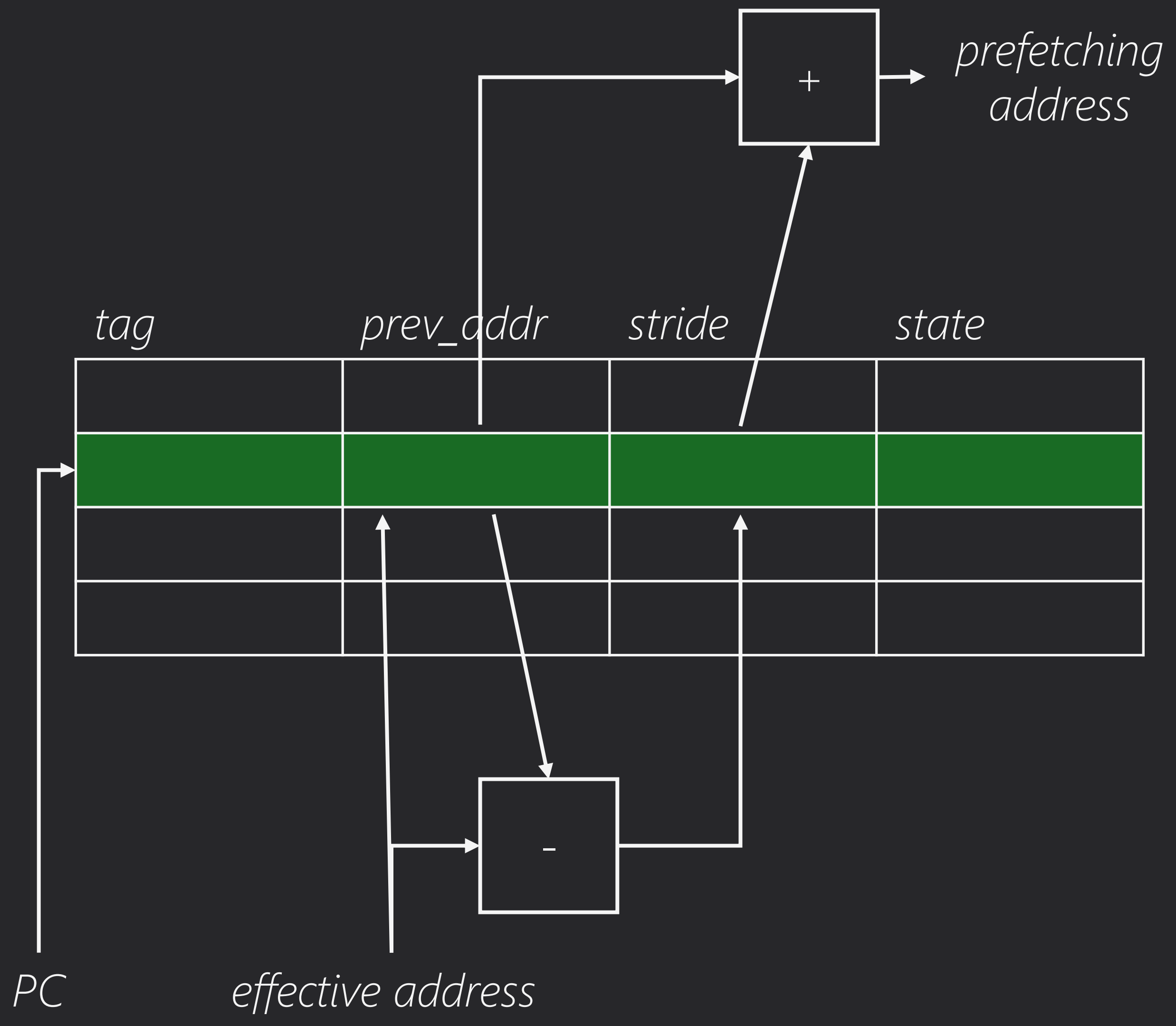
```
int A[100][100];
int B[100][100];
int C[100][100];

for i = 1 to 100:
  for j = 1 to 100:
    for k = 1 to 100:
      A[i][j] += B[i][k] * C[k][j]
```

Address	*Instruction*	*Comment*	*Stride*
500	`LW r4, 0(r2)`	B[i,k]	4B
504	`LW r5, 0(r3)`	C[k,j]	400B
508	`MUL r6, r5, r4`		
512	`LW r7, 0(r1)`	A[i,j]	0B
516	`ADD r7, r7, r6`		
520	`SW r7, 0(r1)`		
524	`ADD r2, r2, 4`		
528	`ADD r3, r3, 400`		
532	`ADD r11, r11, 1`		
536	`BNE r11, r13, 500`		

The reference prediction table (RPT)

- The tag: Address of the load or store instruction
- The prev_addr: The last *effective address*
- The stride: The difference between the last two addresses
- The state: Whether we should make a prediction

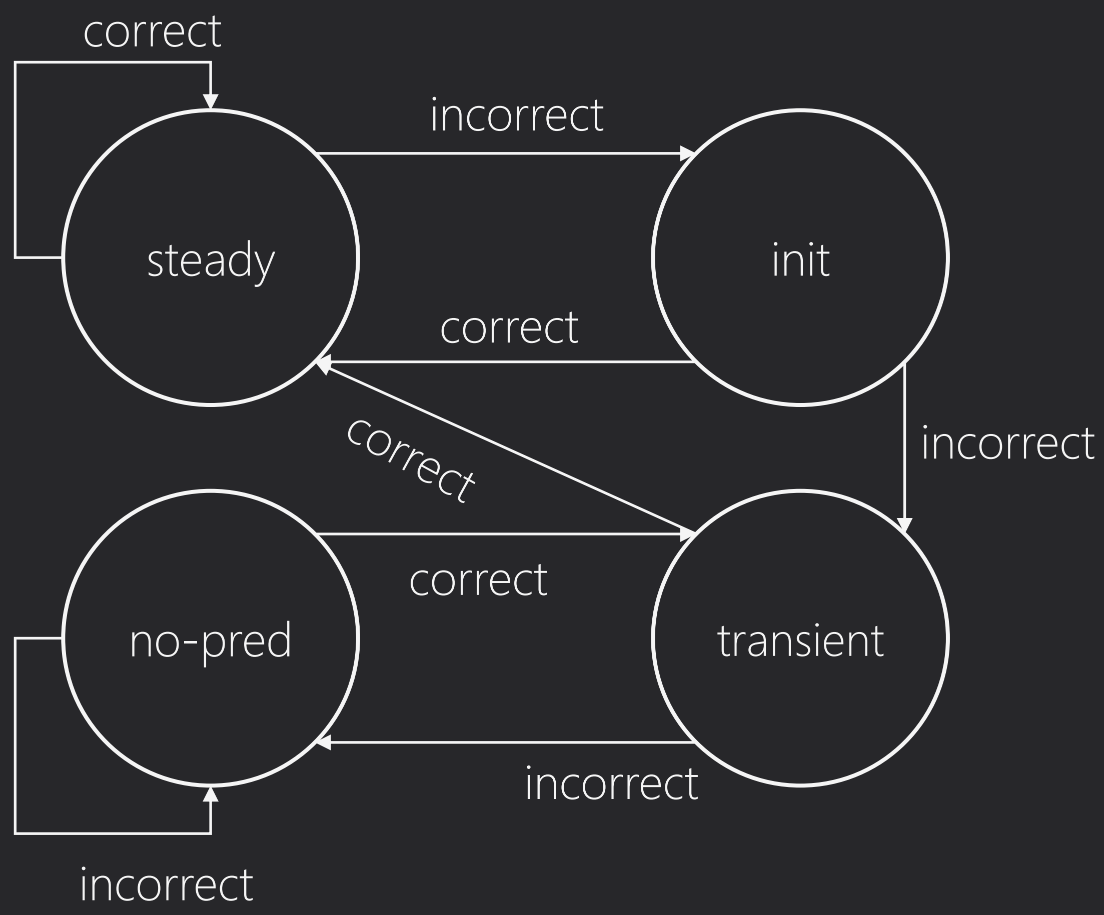
- Mechanism
  1. Record the effective address
  2. Compute the stride
  3. Set the state (compare previous stride with the one just computed)
  4. Predict if the state Is "steady"



State of RPT rows

- Initial
  - e.g., when row is first accessed
- Transient
  - e.g., predictions are not yet stable
- Steady
  - e.g., predictions should be stable
- No prediction
  - e.g., to disable prefetching

Strides are updated on incorrect transitions.



Example: matrix multiply and RPT (0)

Address	*Instruction*	*Comment*	*Stride*
500	`LW r4, 0(r2)`	B[i,k]	4B
504	`LW r5, 0(r3)`	C[k,j]	400B
508	`MUL r6, r5, r4`		
512	`LW r7, 0(r1)`	A[i,j]	0B
516	`ADD r7, r7, r6`		
520	`SW r7, 0(r1)`		
524	`ADD r2, r2, 4`		
528	`ADD r3, r3, 400`		
532	`ADD r11, r11, 1`		
536	`BNE r11, r13, 500`		

Initially empty

tag	*prev_addr*	*stride*	*state*

Example: matrix multiply and RPT (1)

Address	*Instruction*	*Comment*	*Stride*
500	`LW r4, 0(r2)`	B[i,k]	4B
504	`LW r5, 0(r3)`	C[k,j]	400B
508	`MUL r6, r5, r4`		
512	`LW r7, 0(r1)`	A[i,j]	0B
516	`ADD r7, r7, r6`		
520	`SW r7, 0(r1)`		
524	`ADD r2, r2, 4`		
528	`ADD r3, r3, 400`		
532	`ADD r11, r11, 1`		
536	`BNE r11, r13, 500`		

After iteration 1

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

Example: matrix multiply and RPT (2)

Address	*Instruction*	*Comment*	*Stride*
500	`LW r4, 0(r2)`	`B[i,k]`	4B
504	`LW r5, 0(r3)`	`C[k,j]`	400B
508	`MUL r6, r5, r4`		
512	`LW r7, 0(r1)`	`A[i,j]`	0B
516	`ADD r7, r7, r6`		
520	`SW r7, 0(r1)`		
524	`ADD r2, r2, 4`		
528	`ADD r3, r3, 400`		
532	`ADD r11, r11, 1`		
536	`BNE r11, r13, 500`		

After iteration 2

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

Example: matrix multiply and RPT (3)

Address	*Instruction*	*Comment*	*Stride*
500	`LW r4, 0(r2)`	`B[i,k]`	4B
504	`LW r5, 0(r3)`	`C[k,j]`	400B
508	`MUL r6, r5, r4`		
512	`LW r7, 0(r1)`	`A[i,j]`	0B
516	`ADD r7, r7, r6`		
520	`SW r7, 0(r1)`		
524	`ADD r2, r2, 4`		
528	`ADD r3, r3, 400`		
532	`ADD r11, r11, 1`		
536	`BNE r11, r13, 500`		

After iteration 3

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

Analysis of RPT

- Different load and store instructions are separated via their PC
  - Like a cache
- Scalar and zero stride need one transition to steady state
  - From init to steady
- Constant stride need two transitions to steady state
  - From init to transient to steady
- Irregular access patterns don't reach steady state
  - Avoid cache pollution

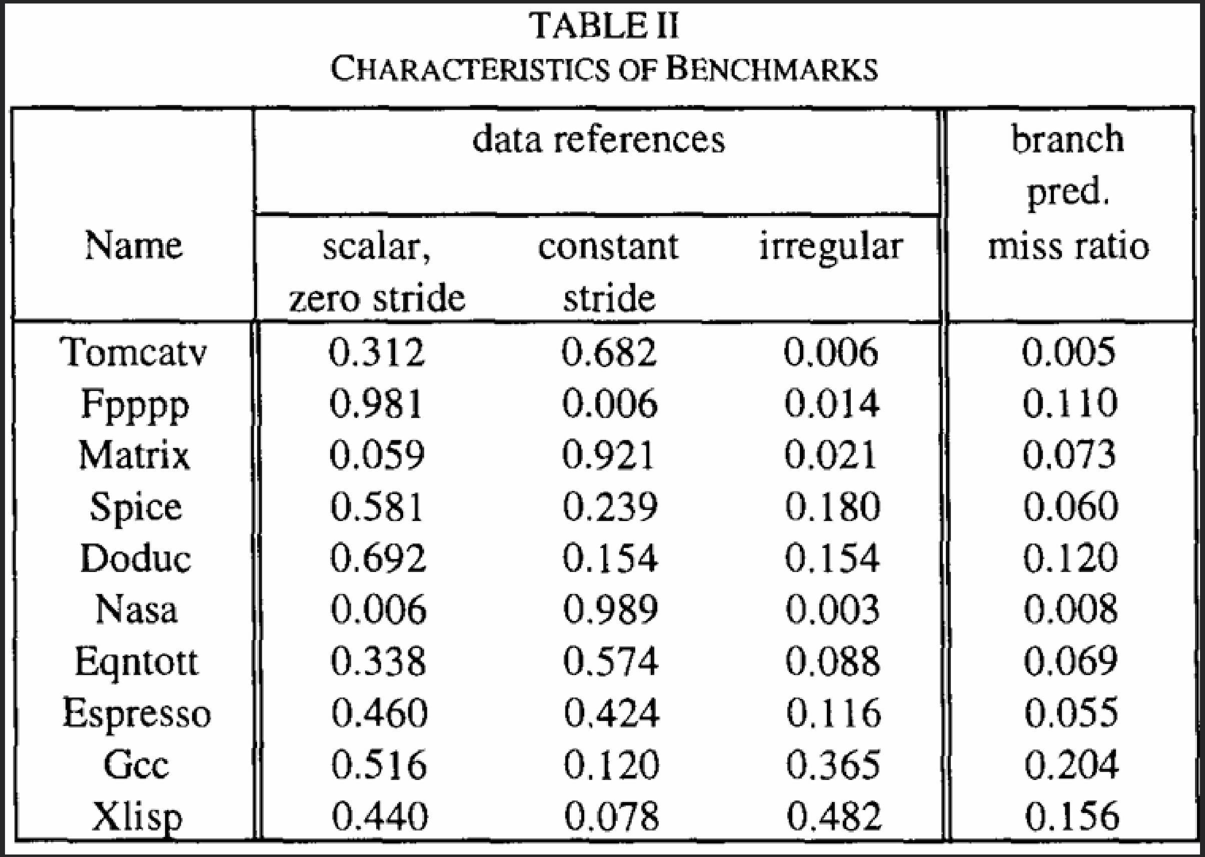


TABLE II
CHARACTERISTICS OF BENCHMARKS

Name	data references			branch pred. miss ratio
	scalar, zero stride	constant stride	irregular	
Tomcatv	0.312	0.682	0.006	0.005
Fpppp	0.981	0.006	0.014	0.110
Matrix	0.059	0.921	0.021	0.073
Spice	0.581	0.239	0.180	0.060
Doduc	0.692	0.154	0.154	0.120
Nasa	0.006	0.989	0.003	0.008
Eqntott	0.338	0.574	0.088	0.069
Espresso	0.460	0.424	0.116	0.055
Gcc	0.516	0.120	0.365	0.204
Xlisp	0.440	0.078	0.482	0.156

Conclusion

- Most modern processors use sequential prefetching for their instruction cache
- Prefetchers can be complex
  - Example shown is stride prefetching
  - Other examples exist (e.g., for pointer-based structures)
- Prefetching can be done in software, too
  - e.g., explicit instructions to initiate a prefetch
  - Programmer needs to ensure prefetches are timely