Lecture 22

Locality Analysis and Prefetching

I. Locality Analysis
   A. Temporal
   B. Spatial
   C. Group
   D. Localized Iteration Space

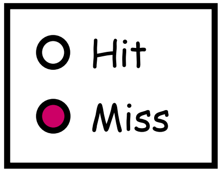
II. Prefetching Pointer-Based Structures

[ALSU 11.5]

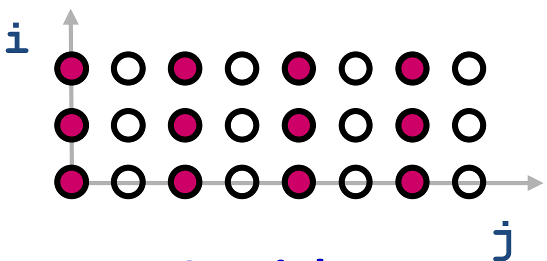
Recall: Types of Data Reuse/Locality

```
double A[3][N], B[N][3];

for i = 0 to 2
  for j = 0 to N-2
    A[i][j] = B[j][0] + B[j+1][0];
```

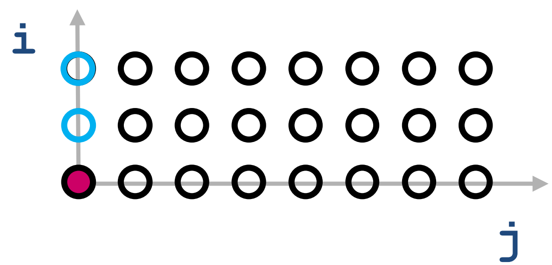


`A[i][j]`



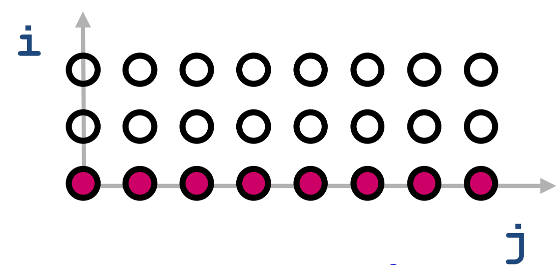
Spatial
(Self)

`B[j][0]`



Temporal
(Group)
except for

`B[j+1][0]`



Temporal
(Self)

(assume row-major, 2 elements per cache line, N small)

I. Predicting Cache Behavior through "Locality Analysis"

- Definitions:
  - Reuse:
    - accessing a location that has been accessed in the past
  - Locality:
    - accessing a location that is now found in the cache

- Key Insights
  - Locality only occurs when there is reuse!
  - BUT, reuse does not necessarily result in locality.
    - why not?

Steps in Locality Analysis

1. Find data reuse
   – if caches were infinitely large, we would be finished
2. Determine "localized iteration space"
   – set of inner loops where the data accessed by an iteration is expected to fit within the cache
3. Find data locality:
   – reuse $\cap$ localized iteration space $\Rightarrow$ locality

Reuse Analysis: Representation

```
for i = 0 to 2
 for j = 0 to N-2
  A[i][j] = B[j][0] + B[j+1][0];
```

- Map *n* loop indices into *d* array indices via array indexing function:

$$\vec{f}(\vec{\imath}) = H\vec{\imath} + \vec{c}$$

$$\texttt{A[i][j]} = \texttt{A}\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right)$$

$$\texttt{B[j][0]} = \texttt{B}\left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right)$$

$$\texttt{B[j+1][0]} = \texttt{B}\left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right)$$

More Complicated Example

```
for i = ...
 for j = 0 to m
  A[2i+2][m-j][i+3j+1] = ...;
```

$$\texttt{A[2i+2][m-j][i+3j+1]} = \texttt{A}\left(\begin{bmatrix} 2 & 0 \\ 0 & -1 \\ 1 & 3 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 2 \\ m \\ 1 \end{bmatrix}\right)$$

Note: Representation is for Affine Array Indexes, i.e.
the index for each dimension of the array is an affine expression of surrounding loop variables and symbolic constants

An expression of one or more variables $x_1, x_2, \ldots, x_n$ is affine if it can be expressed as
$c_0 + c_1x_1 + c_2x_2 + \cdots + c_nx_n$ for constants $c_0, c_1, \ldots, c_n$

A. Finding Temporal Reuse

- Temporal reuse occurs between iterations $\vec{i}_1$ and $\vec{i}_2$ whenever:

$$H\vec{i}_1 + \vec{c} = H\vec{i}_2 + \vec{c}$$
$$H(\vec{i}_1 - \vec{i}_2) = \vec{0}$$

- There is a well-known concept from linear algebra that characterizes when $\vec{i}_1$ and $\vec{i}_2$ satisfy the above equation:
  - Set of all solutions to *H v = 0* is called the *nullspace* of *H*
  - Two iterations refer to the same array element iff the difference of their loop-index vectors is in the nullspace of *H*

- A nullspace can be summarized by its basis vectors
  - Any vector in the nullspace is a linear combination of the basis vectors

Temporal (Self-)Reuse Example

```
for i = 0 to 2
 for j = 0 to 100
  A[i][j] = B[j][0] + B[j+1][0];
```

- For **B[j+1][0]** reuse between iterations (i_1,j_1) and (i_2,j_2) whenever:

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

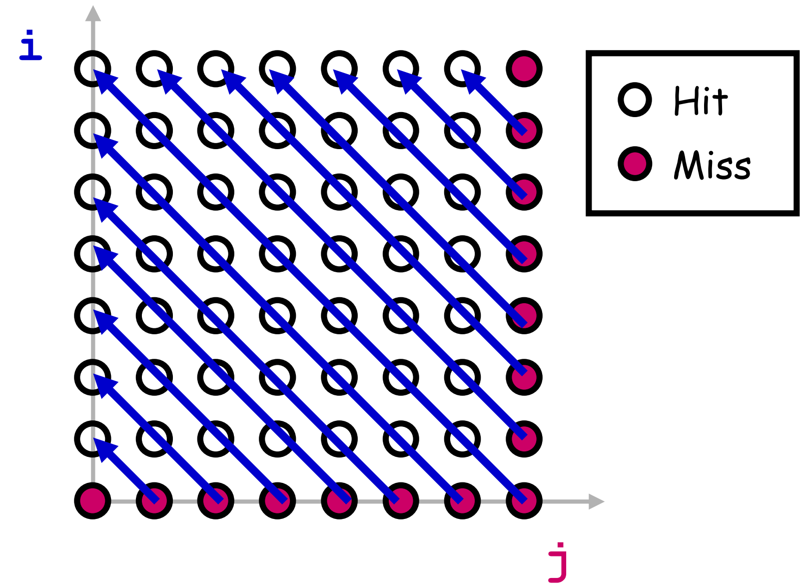
$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- The nullspace of $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ is summarized by the basis vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ because $\mathbf{c}\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ represents all the vectors $\mathbf{v}$ such that $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{v} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
- So reuse occurs whenever $\begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \mathbf{c}\begin{bmatrix} 1 \\ 0 \end{bmatrix}$
  - i.e., whenever $j_1 = j_2$, and regardless of the difference between i_1 and i_2

inner or outer loop?
outer

More Complicated Example

```
for i = 0 to N-1
   for j = 0 to N-1
      A[i+j][0] = i*j;
```



$$\texttt{A[i+j][0]} = \texttt{A}\left(\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right)$$

- Nullspace of $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$ is summarized by the basis vector $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$
- So reuse occurs whenever $\begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \mathbf{c}\begin{bmatrix} 1 \\ -1 \end{bmatrix}$
  - i.e., when $\Delta i = -\Delta j$

B. Computing Spatial Reuse

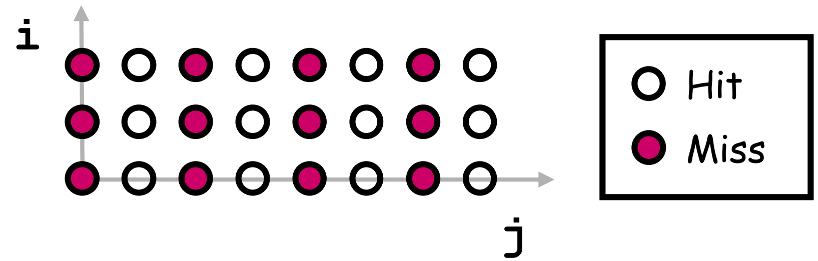
- We assume two array elements share the same cache line iff they differ only in the last dimension
  - E.g., share the same row in a 2-dimensional array
  - Why is this a reasonable approximation? row major order
  - What are its limitations? A row is made up of many cache lines
    Large row could be larger than the cache

- Replace last row of H with zeros, creating H_s
- Find the nullspace of H_s

- Result: vector along which we access the same row

Computing Spatial Reuse: Example

```
for i = 0 to 2
  for j = 0 to 100
    A[i][j] = B[j][0] + B[j+1][0];
```



$$\texttt{A[i][j]} = \texttt{A}\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right)$$

- $H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ $H_s = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$
- Nullspace of H_s is summarized by the basis vector $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$
- So spatial reuse occurs whenever $\begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = c \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
  - i.e., whenever $i_1 = i_2$, and regardless of the difference between j_1 and j_2

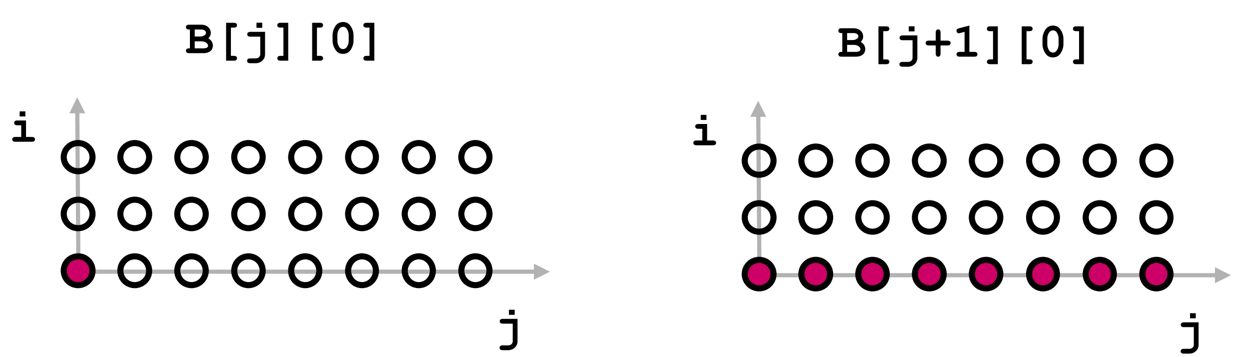
inner or outer loop?
inner

C. Group Reuse (reuse from different static accesses)

```
for i = 0 to 2
 for j = 0 to 100
  A[i][j] = B[j][0] + B[j+1][0];
```

$$H = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

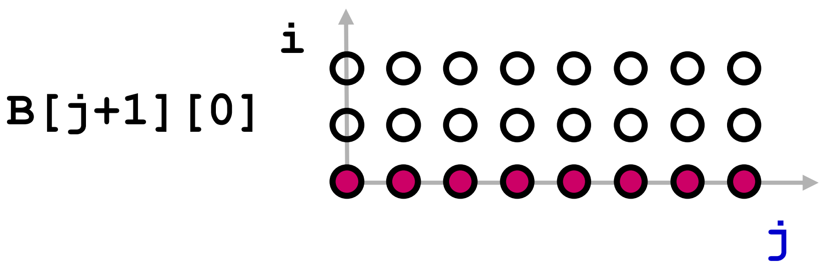
- Limit the analysis to consider only accesses with same H
  - i.e., index expressions that differ only in their constant terms
- Determine when access same location (temporal) or same row (spatial)
- Only the "leading reference" suffers the bulk of the cache misses



D. Localized Iteration Space

- Given finite cache, when does reuse result in locality?
- Localized if accesses less data than *effective cache size*

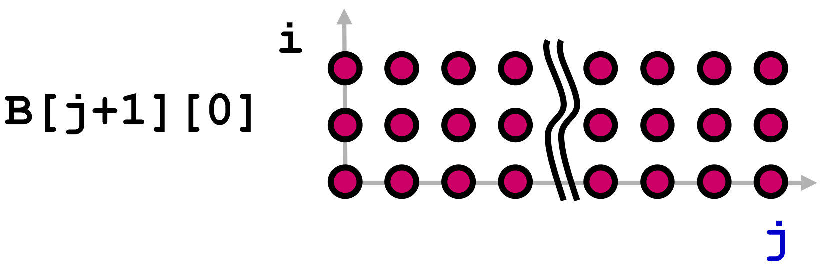
```
for i = 0 to 2
  for j = 0 to 7
    A[i][j] = B[j][0] + B[j+1][0];
```



Localized: both i and j loops

Basis = $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

```
for i = 0 to 2
  for j = 0 to 1000000
    A[i][j] = B[j][0] + B[j+1][0];
```



Localized: j loop only

Basis = $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Computing Locality

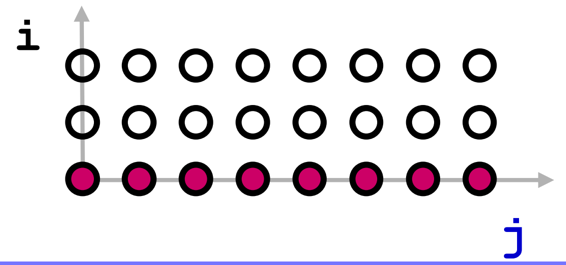
Reuse Vector Space $\cap$ Localized Vector Space $\Rightarrow$ Locality Vector Space

- Example:

```
for i = 0 to 2
  for j = 0 to N-2
    A[i][j] = B[j][0] + B[j+1][0];
```

- If N is small, then both loops are localized:
  - $\text{span}\{\begin{bmatrix}1\\0\end{bmatrix}\} \cap \text{span}\{\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\} \Rightarrow \text{span}\{\begin{bmatrix}1\\0\end{bmatrix}\}$
  - i.e., temporal reuse does result in temporal locality

B[j+1][0]



Computing Locality

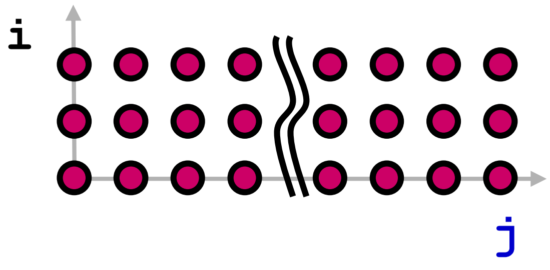
Reuse Vector Space $\cap$ Localized Vector Space $\Rightarrow$ Locality Vector Space

- Example:

```
for i = 0 to 2
  for j = 0 to N-2
    A[i][j] = B[j][0] + B[j+1][0];
```

- If N is large, then only the innermost loop is localized:
  - $\text{span}\{\begin{bmatrix}1\\0\end{bmatrix}\} \cap \text{span}\{\begin{bmatrix}0\\1\end{bmatrix}\} \Rightarrow \text{span}\{\}$
  - i.e., no temporal locality

B[j+1][0]



Locality Analysis Summary

1. Find data reuse
   - Temporal reuse: Compute the nullspace of H
   - Spatial reuse: Compute the nullspace of H_s, which is H with last row zeroed out
   - If caches were infinitely large, we would be finished

2. Determine "localized iteration space"
   - set of inner loops where the data accessed by an iteration is expected to fit within the cache

3. Find data locality:
   - reuse $\cap$ localized iteration space $\Rightarrow$ locality

II. Prefetching

Recall: Compiler Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

Recall: Prefetch Predicate

Locality Type	Miss Instance	Predicate on Iteration Space
None	Every Iteration	True
Temporal	First Iteration	**i** = 0
Spatial	Every L iterations (L elements/cache line)	(**i** mod L) = 0

Example:

```
for i = 0 to 2
 for j = 0 to N-2
  A[i][j] = B[j][0] + B[j+1][0];
```

Reference	Locality	Predicate on Iteration Space
A[i][j]	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$	(**j** mod L) = 0
B[j+1][0]	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$	**i** = 0

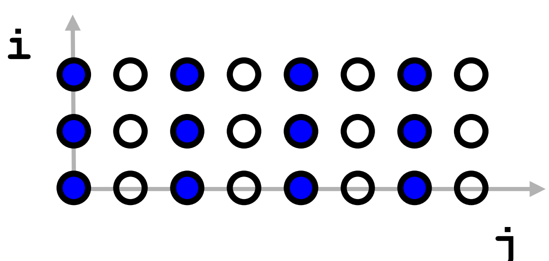
Recall: Loop Splitting for Prefetching Arrays

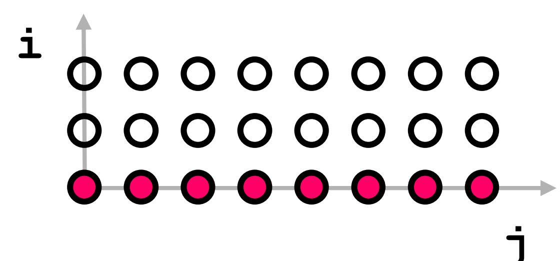
- Decompose loops to isolate cache miss instances
  - cheaper than inserting IF(Prefetch Predicate) statements

Locality Type	**Predicate**	**Loop Transformation**
None	True	None
Temporal	**i** = 0	**Peel** loop **i**
Spatial	(**i** mod L) = 0	**Unroll** loop **i** by L

(L elements/cache line)

Loop peeling: split any problematic first (or last) few iterations from the loop & perform them outside of the loop body





Recall: Example Code with Prefetching

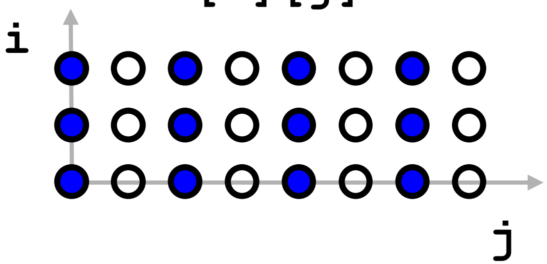
Original Code

```
for (i = 0; i < 3; i++)
  for (j = 0; j < 100; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```

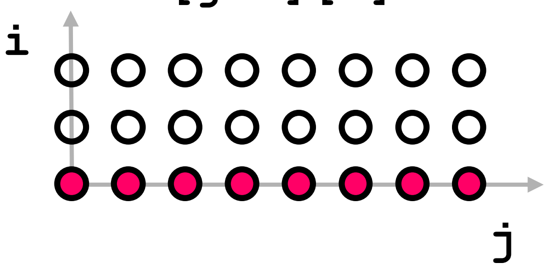
Cache Hit

Cache Miss

A[i][j]



B[j+1][0]



i = 0

```
prefetch(&B[0][0]);
for (j = 0; j < 6; j += 2) {
  prefetch(&B[j+1][0]);
  prefetch(&B[j+2][0]);
  prefetch(&A[0][j]);
}
for (j = 0; j < 94; j += 2) {
  prefetch(&B[j+7][0]);
  prefetch(&B[j+8][0]);
  prefetch(&A[0][j+6]);
  A[0][j] = B[j][0]+B[j+1][0];
  A[0][j+1] = B[j+1][0]+B[j+2][0];
}
for (j = 94; j < 100; j += 2) {
  A[0][j] = B[j][0]+B[j+1][0];
  A[0][j+1] = B[j+1][0]+B[j+2][0];
}
```

i > 0

```
for (i = 1; i < 3; i++) {
  for (j = 0; j < 6; j += 2)
    prefetch(&A[i][j]);
  for (j = 0; j < 94; j += 2) {
    prefetch(&A[i][j+6]);
    A[i][j] = B[j][0] + B[j+1][0];
    A[i][j+1] = B[j+1][0] + B[j+2][0];
  }
  for (j = 94; j < 100; j += 2) {
    A[i][j] = B[j][0] + B[j+1][0];
    A[i][j+1] = B[j+1][0] + B[j+2][0];
  }
}
```

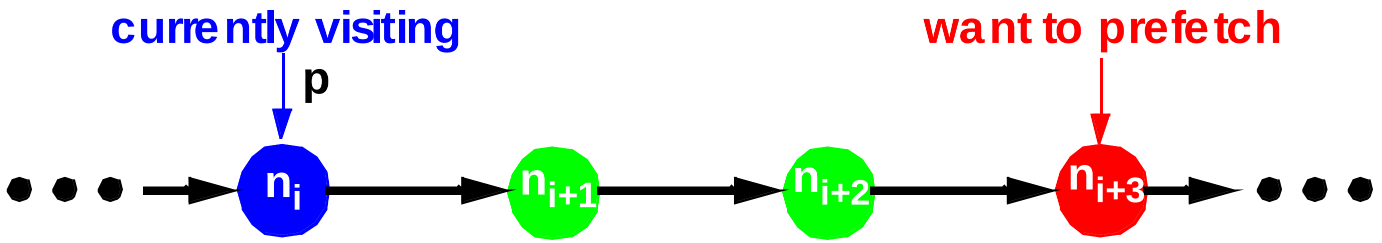
Today: Prefetching for Pointer-Based Structures

- Examples:
  - linked lists, trees, graphs, ...
- A common method of building large data structures
  - especially in non-numeric programs
- Cache miss behavior is a concern because:
  - large data set with respect to the cache size
  - temporal locality may be poor
  - little spatial locality among consecutively-accessed nodes

Goal:

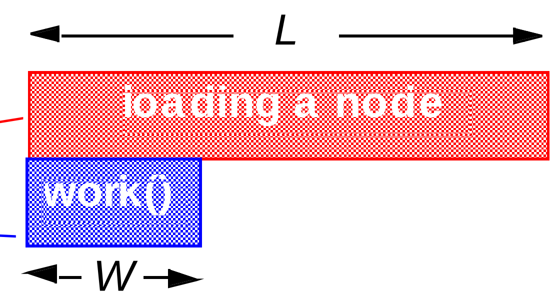
- Automatic compiler-based prefetching for pointer-based data structures

Scheduling Prefetches for Pointer-Based Data Structures




```
p = &n0
while (p){
    work(p->data);
    p = p->next;
}
```

load *p here

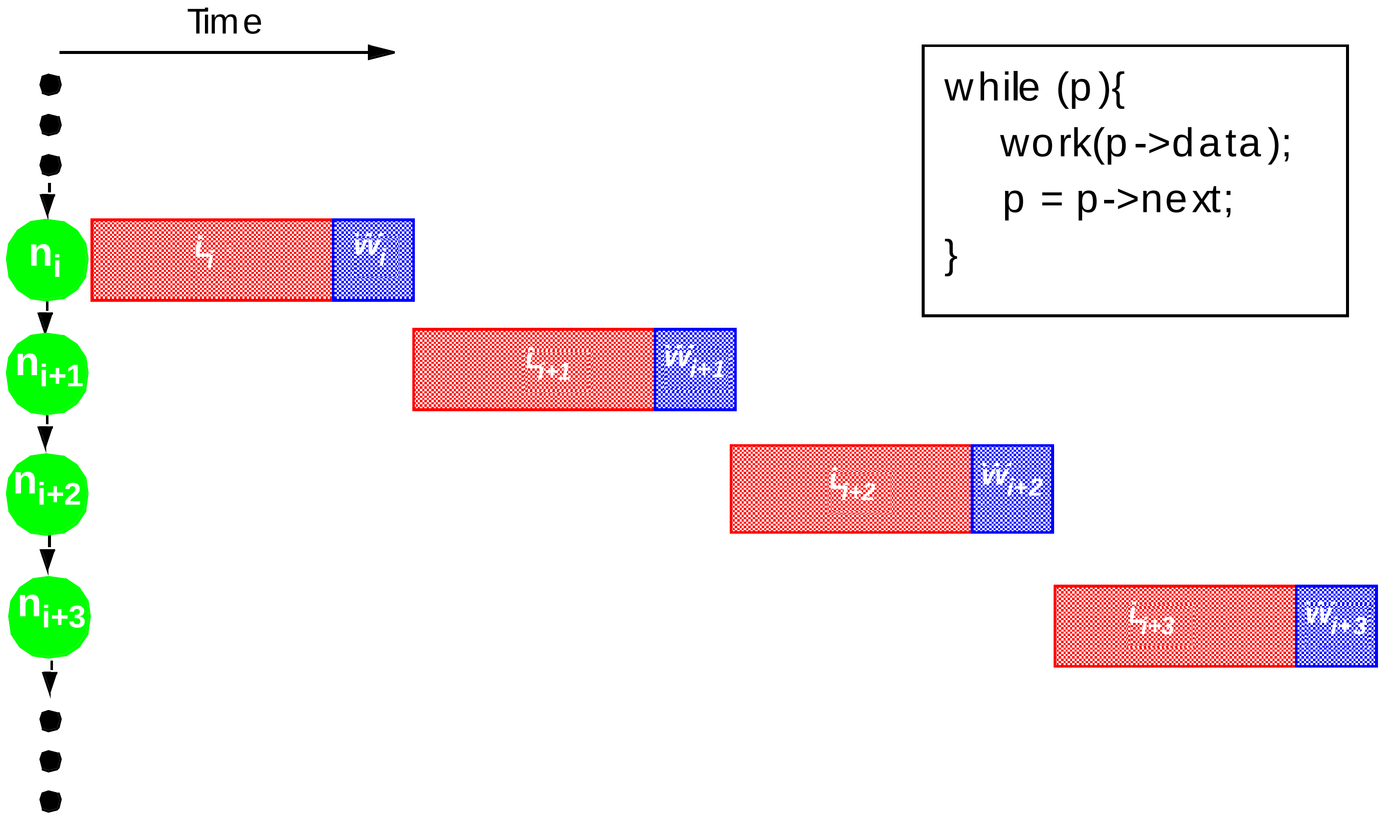


Our Goal: *fully hide latency*

– thus achieving fastest possible computation rate of 1/W

- e.g., if L = 3W, we must prefetch 3 nodes ahead to achieve this

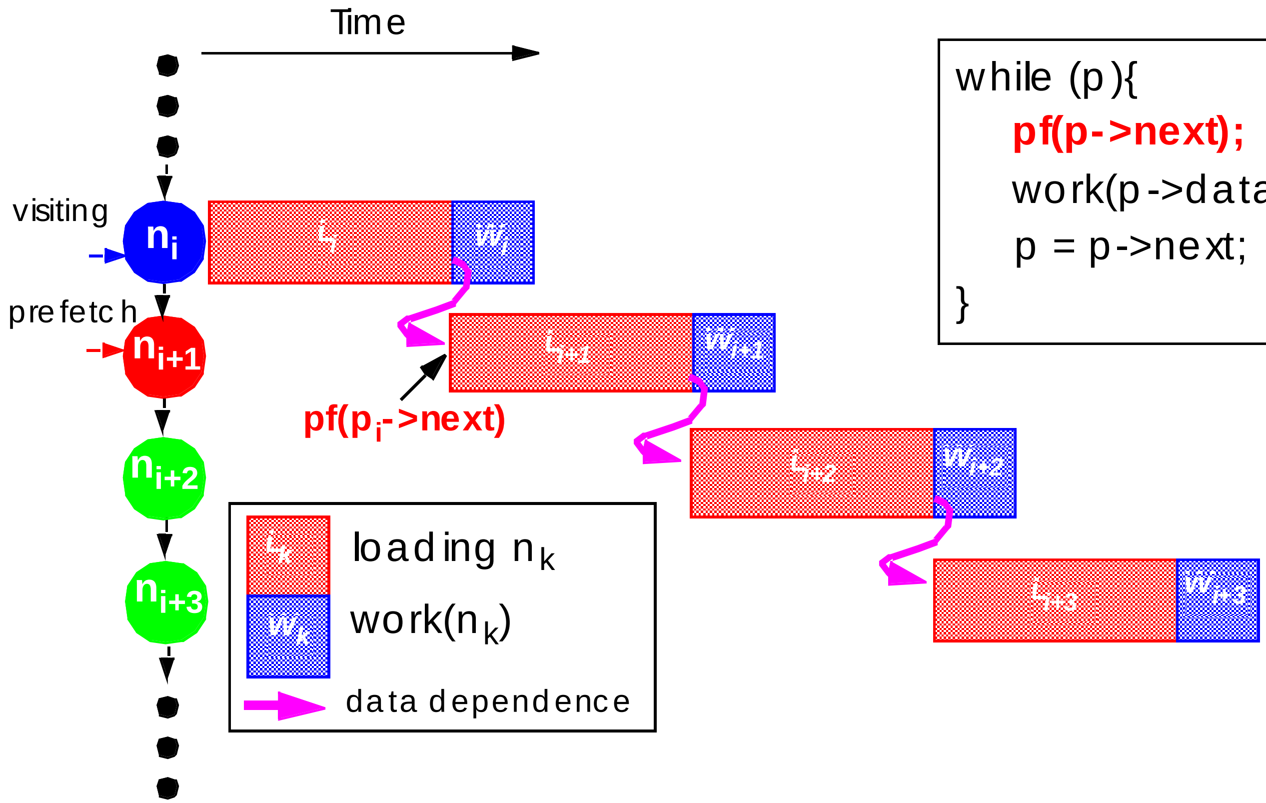
Performance without Prefetching




```
while (p){
    work(p->data);
    p = p->next;
}
```

computation rate = 1 / (L+W)

Prefetching One Node Ahead



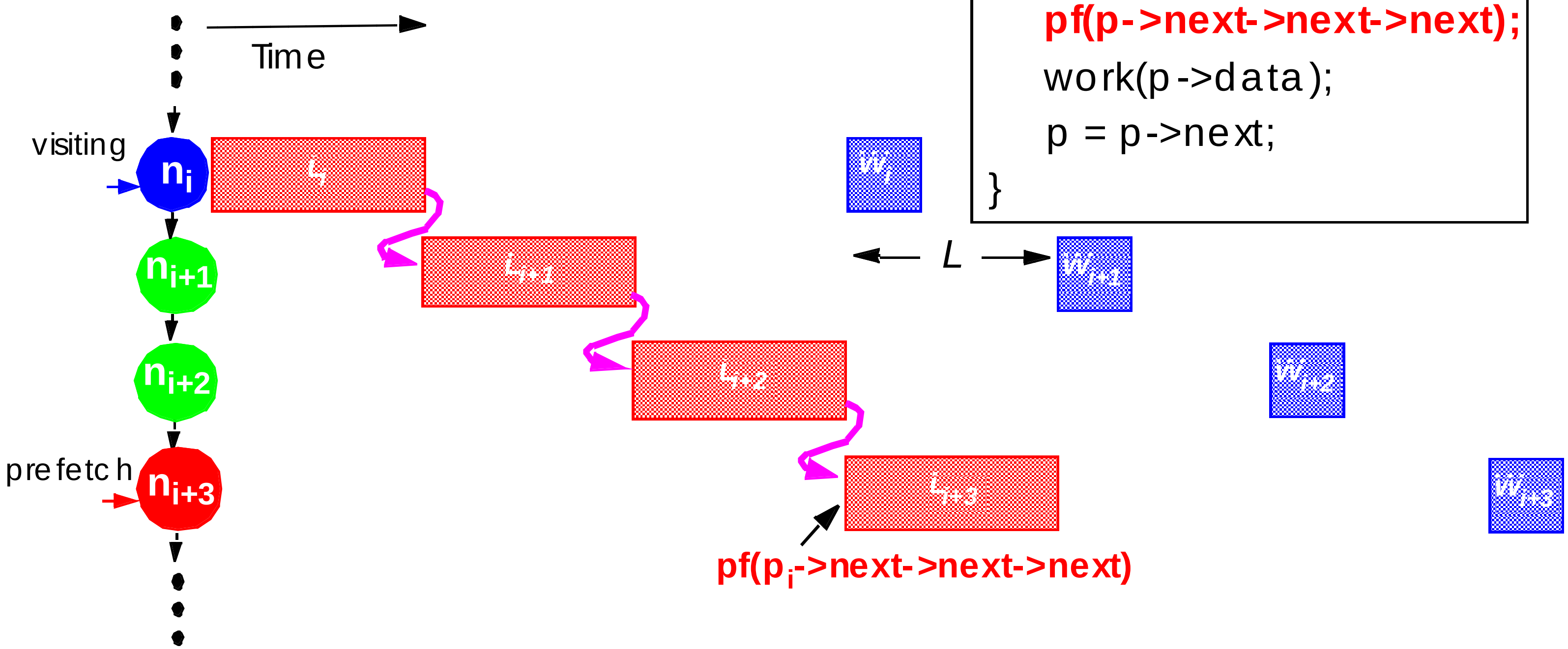

```
while (p){
    pf(p->next);
    work(p->data);
    p = p->next;
}
```

- Computation is overlapped with memory accesses

computation rate = 1/L

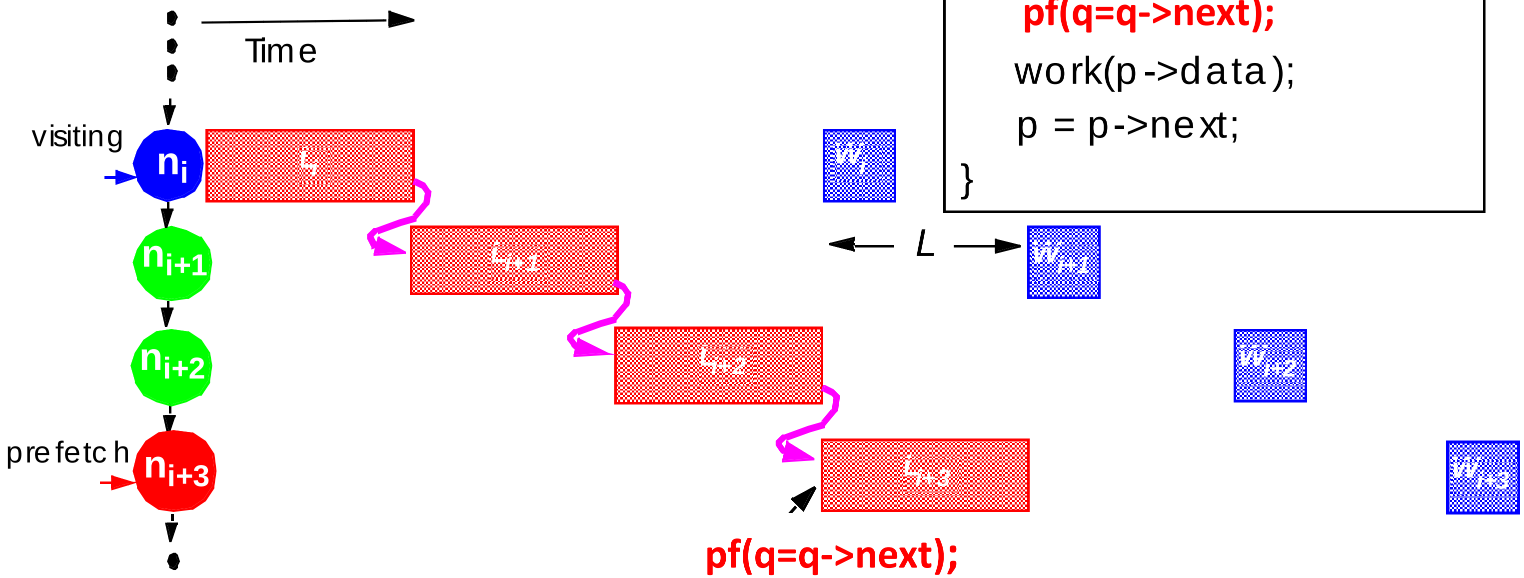
Prefetching Three Nodes Ahead

```
while (p){
    pf(p->next->next->next);
    work(p->data);
    p = p->next;
}
```



Prefetching Three Nodes Ahead

```
q=p->next->next;
while(q) {
    pf(q=q->next);
    work(p->data);
    p = p->next;
}
```

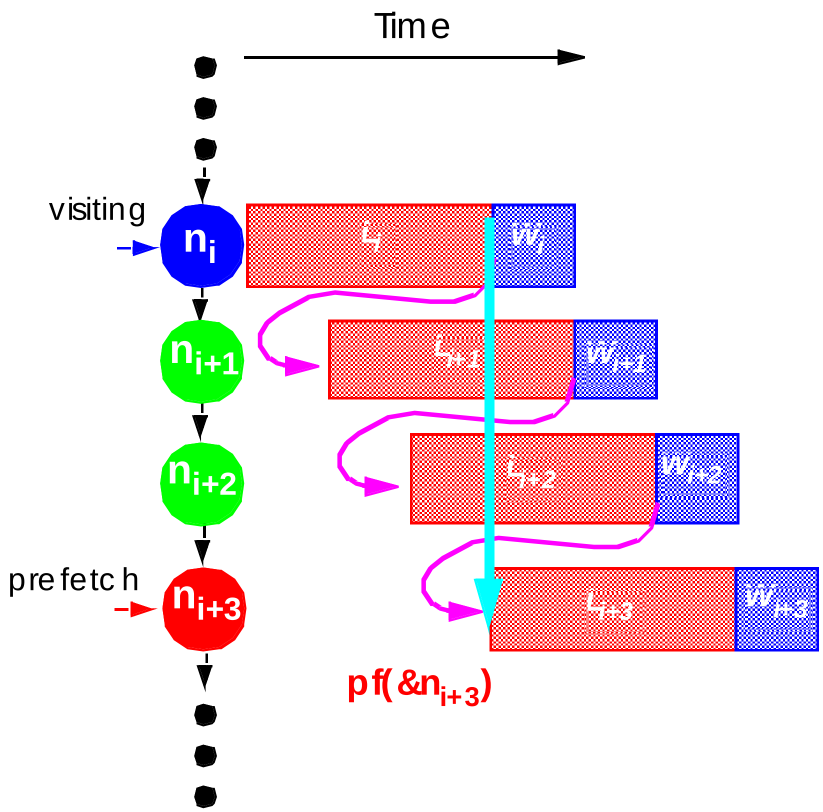


computation rate does not improve (still = 1/L)!

Pointer-Chasing Problem:

- any scheme which follows the pointer chain is limited to a rate of 1/L

Our Goal: Fully Hide Latency




```
while (p){
    pf(&n_{i+3});
    work(p->data);
    p = p->next;
}
```

- achieves the fastest possible computation rate of 1/W

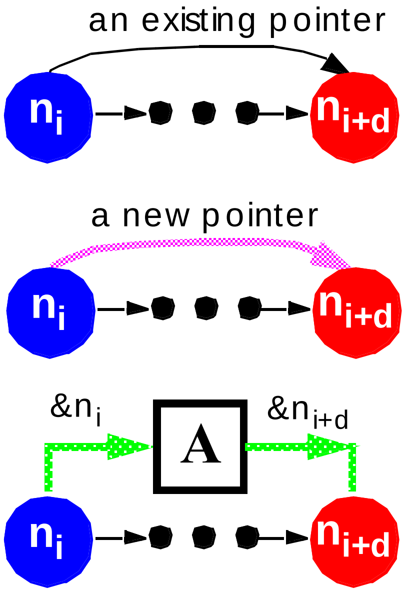
Overcoming the Pointer-Chasing Problem

Key:

- n_i needs to know $\&n_{i+d}$ without referencing the d-1 intermediate nodes

Three Algorithms:

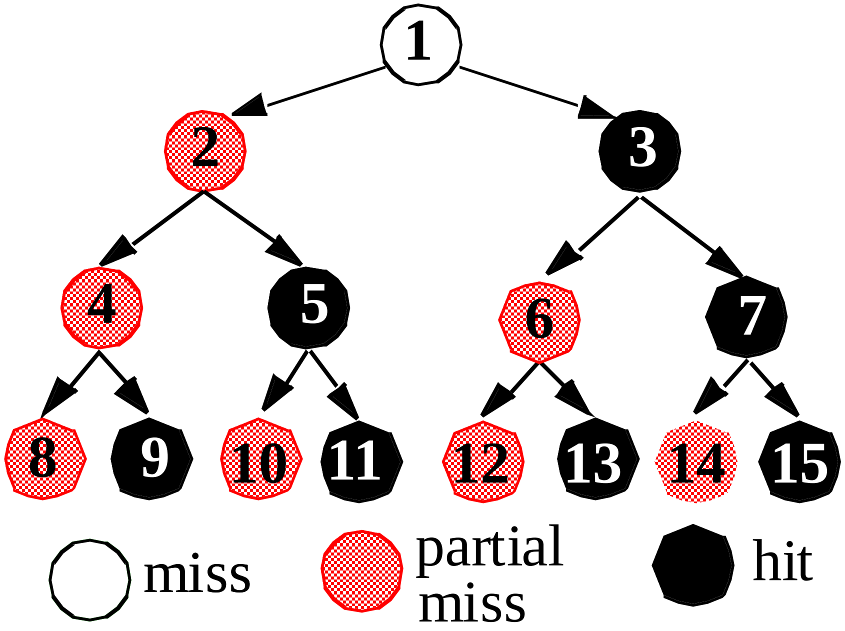
- use *existing* pointer(s) in n_i to approximate $\&n_{i+d}$
  - Greedy Prefetching
- add *new* pointer(s) to n_i to approximate $\&n_{i+d}$
  - History-Pointer Prefetching
- compute $\&n_{i+d}$ *directly* from $\&n_i$ (no ptr deref)
  - Data-Linearization Prefetching



Greedy Prefetching

- Prefetch all neighboring nodes (simplified definition)
  - only one will be followed by the immediate control flow
  - hopefully, we will visit other neighbors later

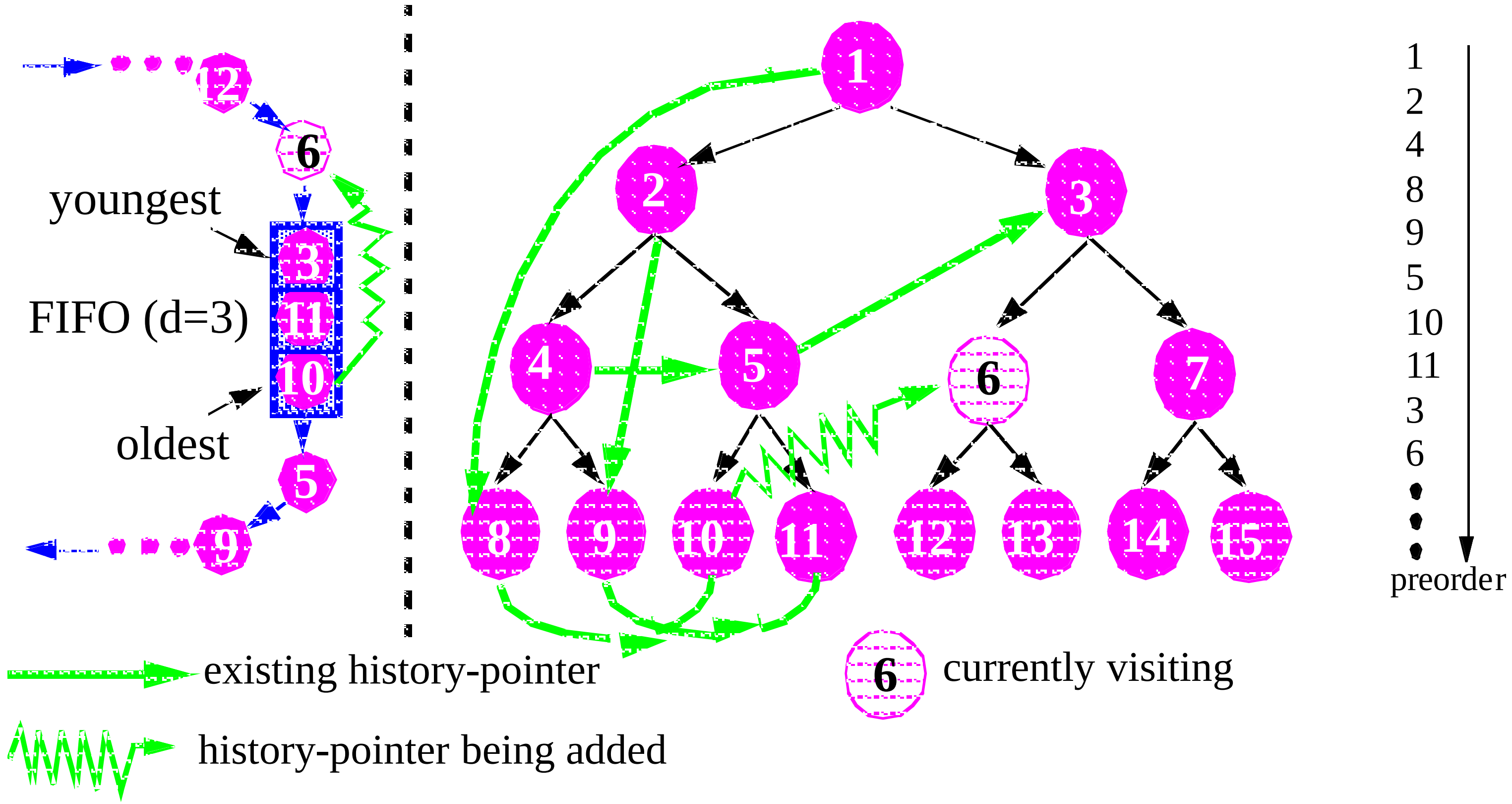
```
preorder(treeNode * t){
  if (t != NULL){
    pf(t->left);
    pf(t->right);
    process(t->data);
    preorder(t->left);
    preorder(t->right);
  }
}
```



- Reasonably effective in practice
- However, little control over the prefetching distance

History-Pointer Prefetching

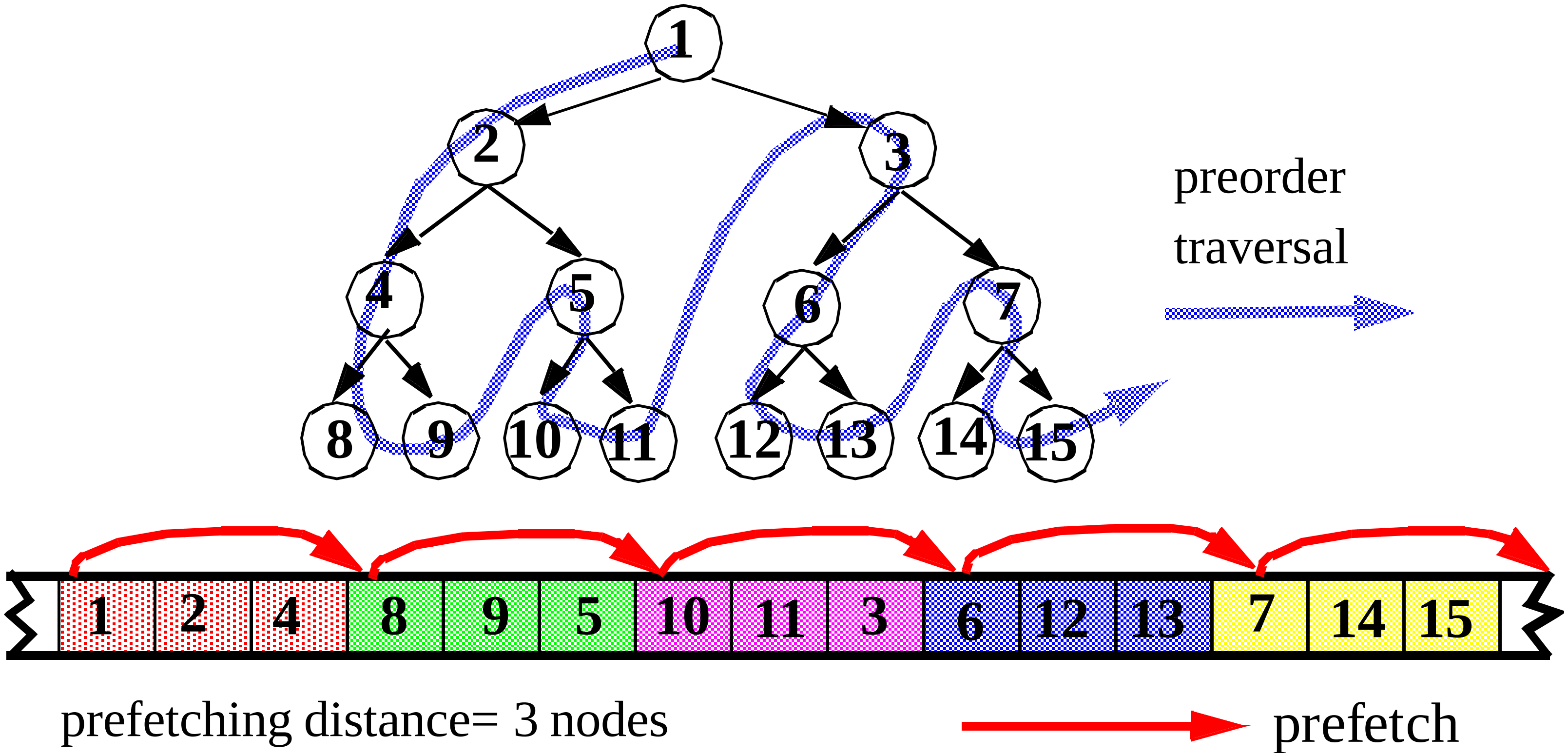
- Add new pointer(s) to each node
  - history-pointers are obtained from some recent traversal



- Trade space & time for better control over prefetching distances

Data-Linearization Prefetching

- No pointer dereferences are required
- Map nodes close in the traversal to contiguous memory



Summary of Prefetching Algorithms for Pointer Structures

	Greedy	History-Pointer	Data-Linearization
Control over Prefetching Distance			
Applicability to Pointer-Based Data Structures			
Overhead in Preparing Prefetch Addresses			
Ease of Implementation			

Summary of Prefetching Algorithms for Pointer Structures

	Greedy	History-Pointer	Data-Linearization
Control over Prefetching Distance	little	more precise	more precise
Applicability to Pointer-Based Data Structures	any	revisited; changes only slowly	must have a major traversal order; changes only slowly
Overhead in Preparing Prefetch Addresses	none	space + time	none in practice
Ease of Implementation	relatively straightforward	more difficult	more difficulty

- Greedy prefetching is the most widely applicable algorithm

Today's Class: Locality Analysis and Prefetching

I. Locality Analysis
   A. Temporal
   B. Spatial
   C. Group
   D. Localized Iteration Space

II. Prefetching Pointer-Based Structures

Friday's Class

- Array Dependence Analysis & Parallelization
  - ALSU 11.6