# CSC D70:
# Compiler Optimization
# Prefetching

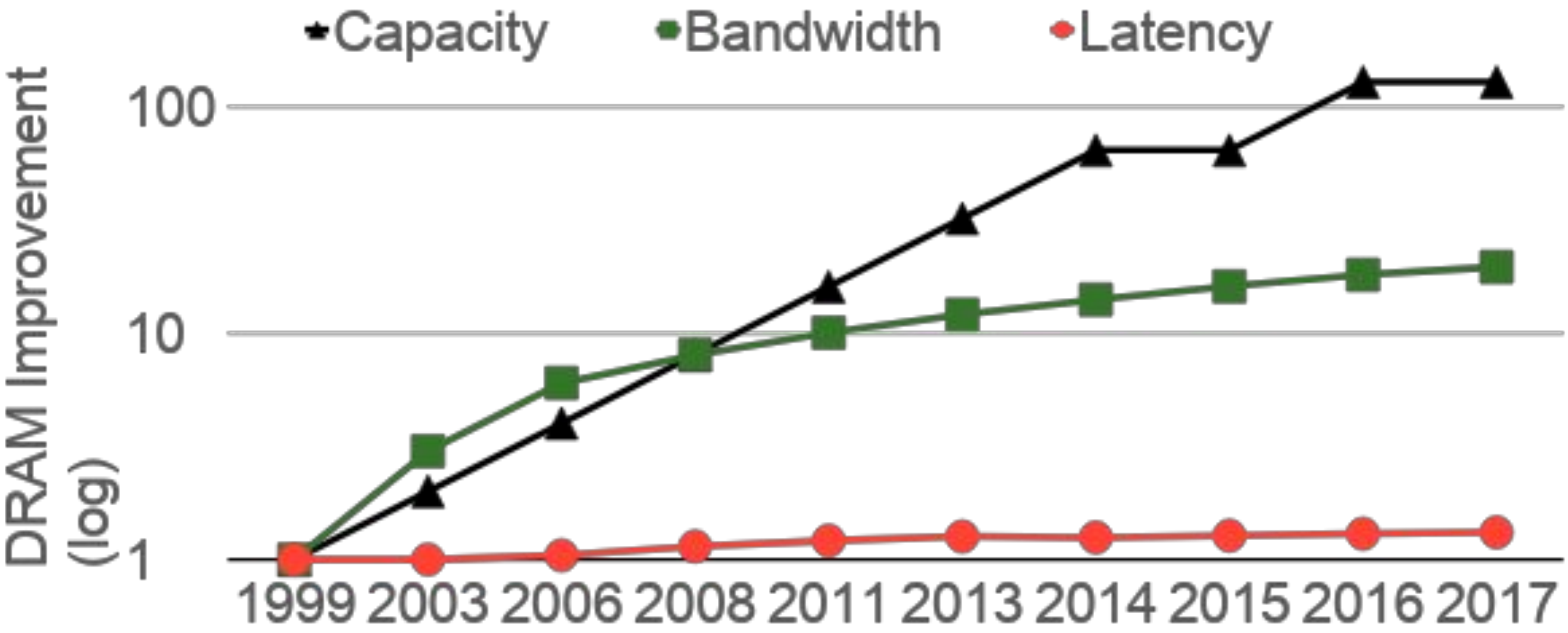
Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons*

# The Memory Latency Problem



- ▯ processor speed >> ▯ memory speed
- caches are not a panacea

# Prefetching for Arrays: Overview

- Tolerating Memory Latency
- Prefetching Compiler Algorithm and Results
- Implications of These Results

# Coping with Memory Latency

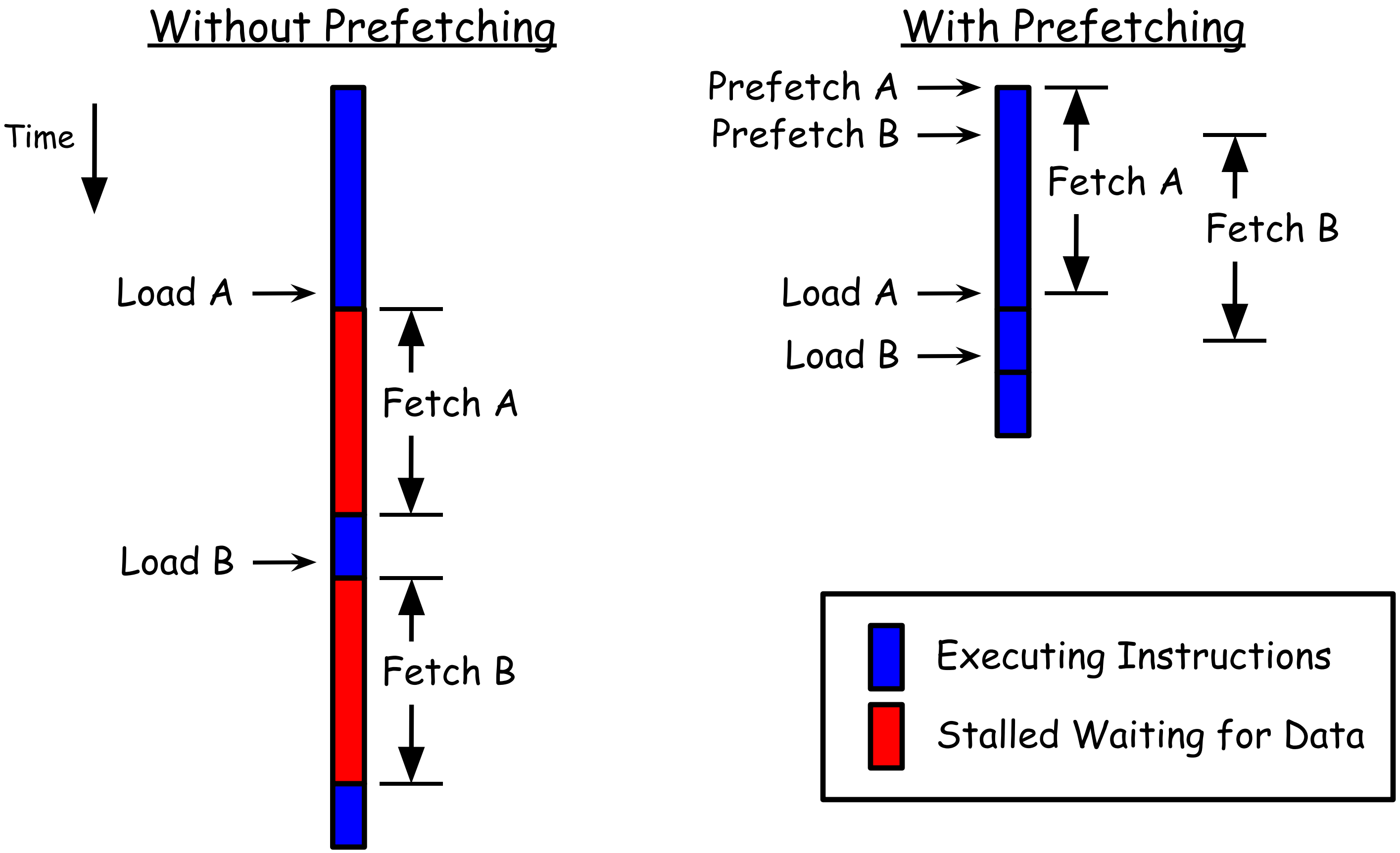
**Reduce Latency:**

- Locality Optimizations
  - reorder iterations to improve cache reuse

**Tolerate Latency:**

- Prefetching
  - move data close to the processor before it is needed

# Tolerating Latency Through Prefetching



- overlap memory accesses with computation and other accesses

# Types of Prefetching

Cache Blocks:

- (-) limited to unit-stride accesses

Nonblocking Loads:

- (-) limited ability to move back before use

Hardware-Controlled Prefetching:

- (-) limited to constant-strides and by branch prediction
- (+) no instruction overhead

Software-Controlled Prefetching:

- (-) software sophistication and overhead
- (+) minimal hardware support and broader coverage

# Prefetching Goals

- Domain of Applicability
- Performance Improvement
  - maximize benefit
  - minimize overhead

# Prefetching Concepts

*possible* only if addresses can be determined ahead of time
*coverage factor* = fraction of misses that are prefetched
*unnecessary* if data is already in the cache
*effective* if data is in the cache when later referenced

Analysis: what to prefetch

- maximize coverage factor
- minimize unnecessary prefetches

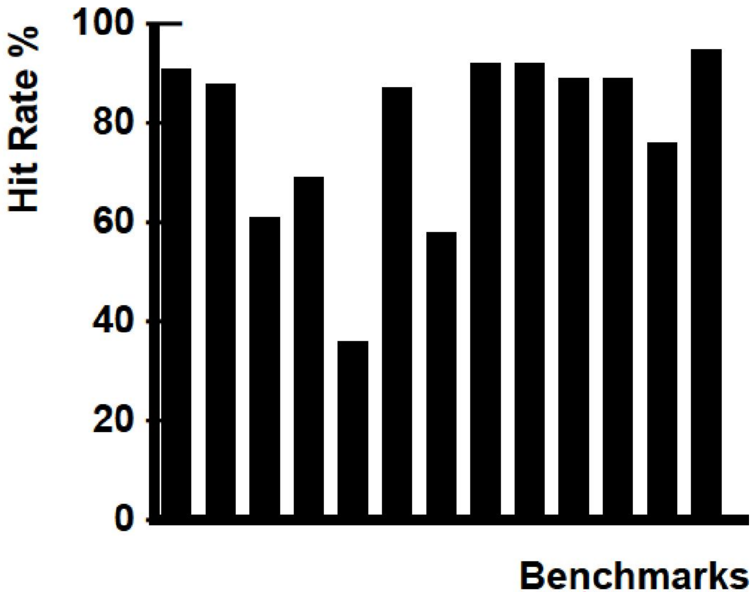
Scheduling: when/how to schedule prefetches

- maximize effectiveness
- minimize overhead per prefetch

# Reducing Prefetching Overhead

- instructions to issue prefetches
- extra demands on memory system

Hit Rates for Array Accesses



- important to minimize unnecessary prefetches

# Compiler Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

# Steps in Locality Analysis

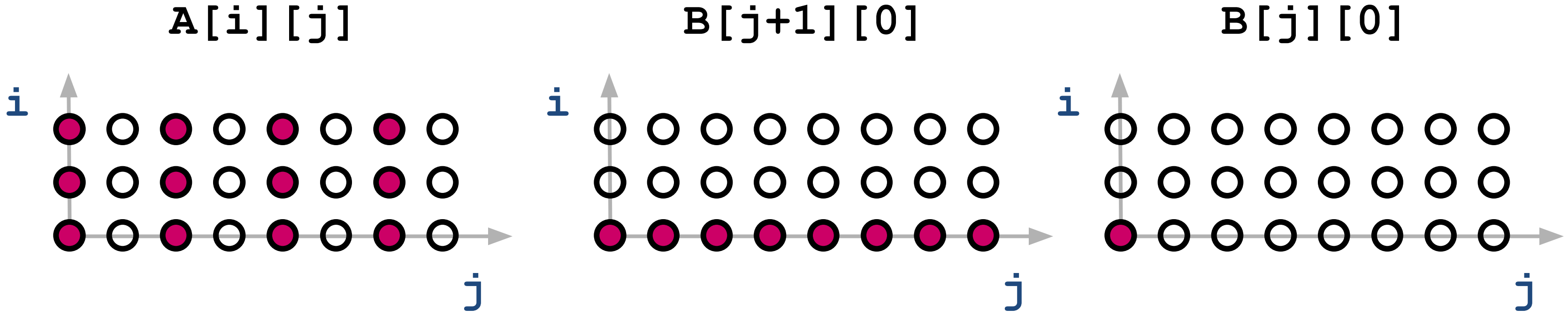
1. Find data reuse
   - if caches were infinitely large, we would be finished
2. Determine “localized iteration space”
   - set of inner loops where the data accessed by an iteration is expected to fit within the cache
3. Find data locality:
   - reuse ∩ localized iteration space ⇒ locality

# Data Locality Example

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

○ Hit

● Miss



Spatial

Temporal

Group

# Reuse Analysis: Representation

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

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

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

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

# Finding Temporal Reuse

- Temporal reuse occurs between iterations $\vec{\imath}1$ and $\vec{\imath}2$ whenever:

$$H\vec{\imath}_1 + \vec{c} = H\vec{\imath}_2 + \vec{c}$$

$$H(\vec{\imath}_1 - \vec{\imath}_2) = \vec{0}$$

- Rather than worrying about individual values $\vec{\imath}_1$ of $\vec{\imath}_2$ and, we say that reuse occurs along direction $\vec{r}$ vector when:

$$H(\vec{r}) = \vec{0}$$

- Solution: compute the *nullspace* of *H*

# Temporal Reuse Example

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

- Reuse between iterations $(i_1,j_1)$ and $(i_2,j_2)$ whenever:

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

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

- True whenever $j_1 = j_2$, and regardless of the difference between $i_1$ and $i_2$.
  - i.e. whenever the difference lies along the nullspace of $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
  - which is span{(1,0)} (i.e. the outer loop).

# Prefetch Predicate

| Locality Type | Miss Instance | Predicate |
|---|---|---|
| None | Every Iteration | True |
| Temporal | First Iteration | $i = 0$ |
| Spatial | Every l iterations<br>(l = cache line size) | $(i \bmod l) = 0$ |

Example:

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

| Reference | Locality | Predicate |
|---|---|---|
| `A[i][j]` | $\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$ | $(j \bmod 2) = 0$ |
| `B[j+1][0]` | $\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$ | $i = 0$ |

# Compiler Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

# Loop Splitting

- Decompose loops to isolate cache miss instances
  - cheaper than inserting IF 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 |

- Apply transformations recursively for nested loops
- Suppress transformations when loops become too large
  - avoid code explosion

# Software Pipelining

$$\text{Iterations Ahead} = \left\lceil \frac{l}{s} \right\rceil$$

where $l$ = memory latency, $s$ = shortest path through loop body

Original Loop

```
for (i = 0; i<100; i++)
   a[i] = 0;
```

Software Pipelined Loop
(5 iterations ahead)

```
for (i = 0; i<5; i++)        /* Prolog */
   prefetch(&a[i]);

for (i = 0; i<95; i++) {  /* Steady State*/
   prefetch(&a[i+5]);
   a[i] = 0;
}

for (i = 95; i<100; i++)  /* Epilog */
   a[i] = 0;
```

# Example Revisited

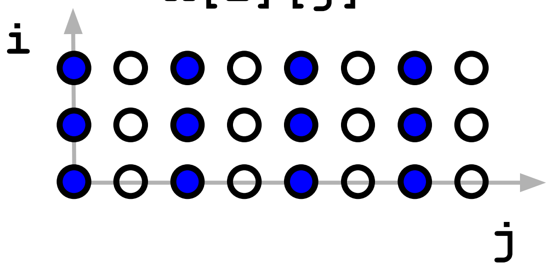
## Original Code

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

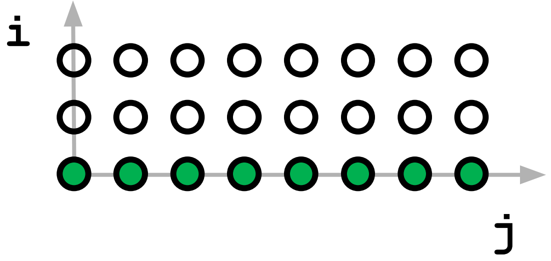
○ Cache Hit

● ● Cache Miss

**A[i][j]**



**B[j+1][0]**



## Code with Prefetching

i = 0

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

# Prefetching Indirections

```
for (i = 0; i<100; i++)
   sum += A[index[i]];
```

Analysis: what to prefetch

- both dense and indirect references
- difficult to predict whether indirections hit or miss

Scheduling: when/how to issue prefetches

- modification of software pipelining algorithm

# Software Pipelining for Indirections

Original Loop

```
for (i = 0; i<100; i++)
   sum += A[index[i]];
```

Software Pipelined Loop
(5 iterations ahead)

```
for (i = 0; i<5; i++)          /* Prolog 1 */
   prefetch(&index[i]);

for (i = 0; i<5; i++) {        /* Prolog 2 */
   prefetch(&index[i+5]);
   prefetch(&A[index[i]]);
}
for (i = 0; i<90; i++) {  /* Steady State*/
   prefetch(&index[i+10]);
   prefetch(&A[index[i+5]]);
   sum += A[index[i]];
}
for (i = 90; i<95; i++) {  /* Epilog 1 */
   prefetch(&A[index[i+5]]);
   sum += A[index[i]];
}
for (i = 95; i<100; i++)   /* Epilog 2 */
   sum += A[index[i]];
```

# Summary of Results

**Dense Matrix Code:**

- eliminated 50% to 90% of memory stall time
- overheads remain low due to prefetching selectively
- significant improvements in overall performance (6 over 45%)

**Indirections, Sparse Matrix Code:**

- expanded coverage to handle some important cases

# Prefetching for Arrays: Concluding Remarks

- Demonstrated that software prefetching is effective
  - selective prefetching to eliminate overhead
  - dense matrices and indirections / sparse matrices
  - uniprocessors and multiprocessors

- Hardware should focus on providing sufficient memory bandwidth

# Prefetching for Recursive Data Structures

# Recursive Data 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

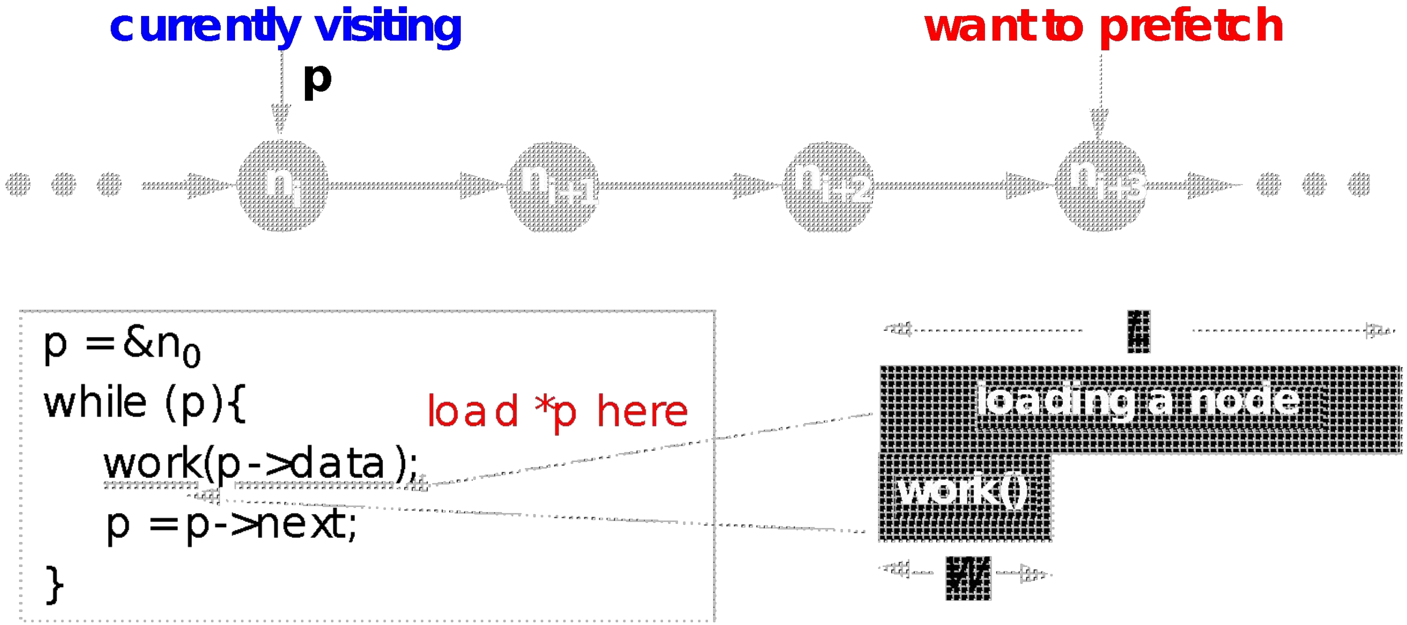
Goal:

- Automatic Compiler-Based Prefetching for Recursive Data Structures

# Overview

- Challenges in Prefetching Recursive Data Structures
- Three Prefetching Algorithms
- Experimental Results
- Conclusions

# Scheduling Prefetches for Recursive Data Structures



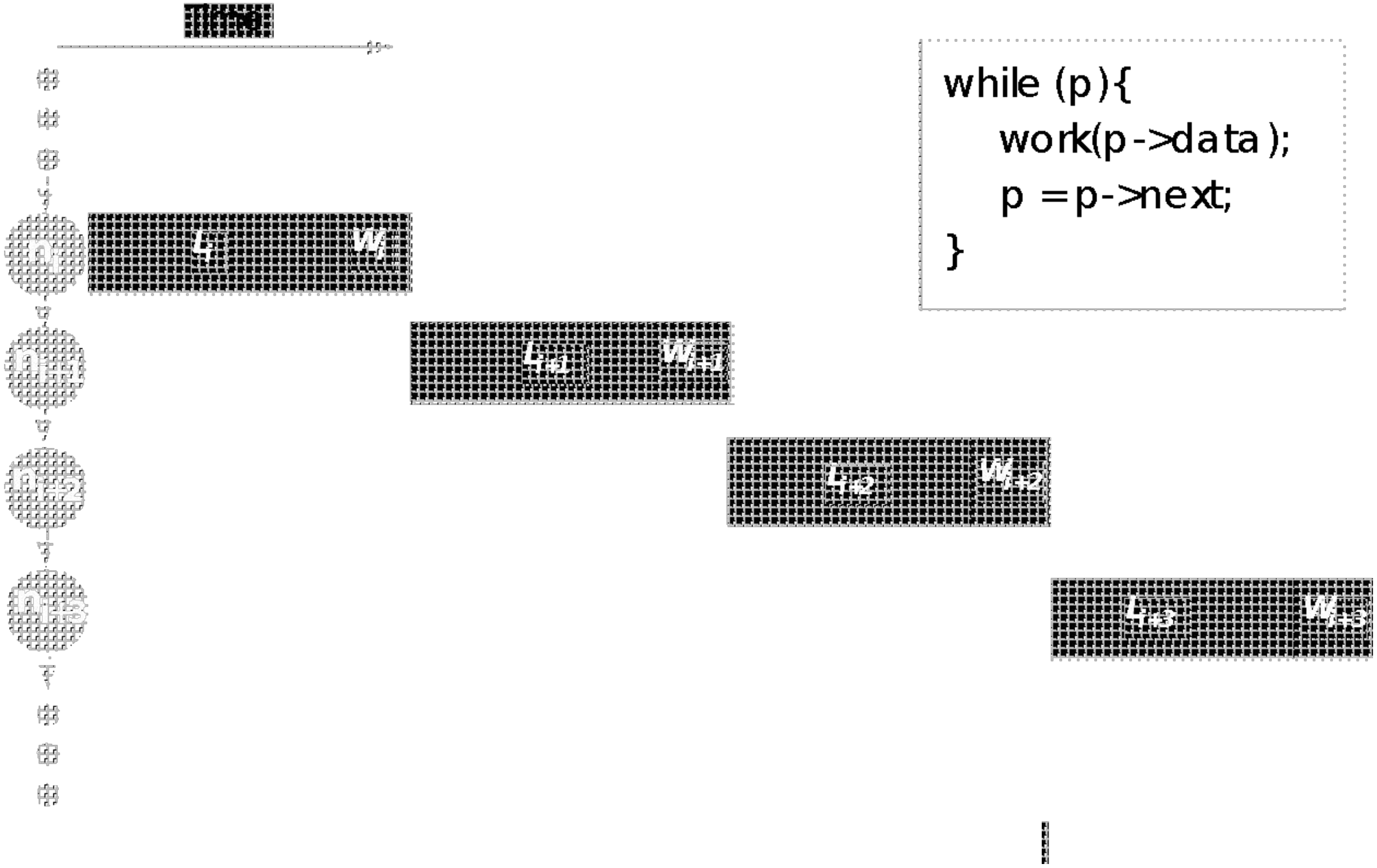
```
p = &n0
while (p){
    work(p->data);
    p = p->next;
}
```

<u>Our Goal</u>: *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

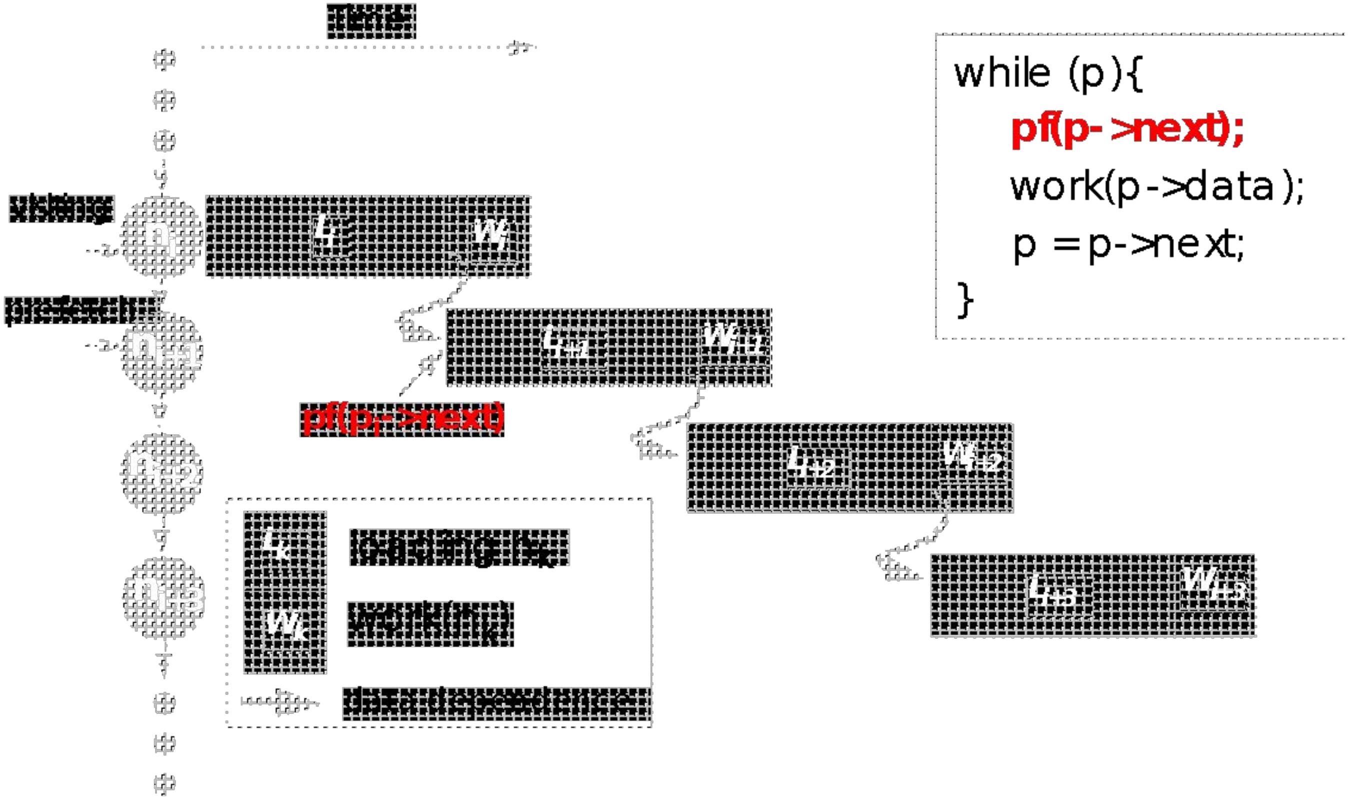
# Performance without Prefetching



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

computation rate = 1 / (L+W)

# Prefetching One Node Ahead

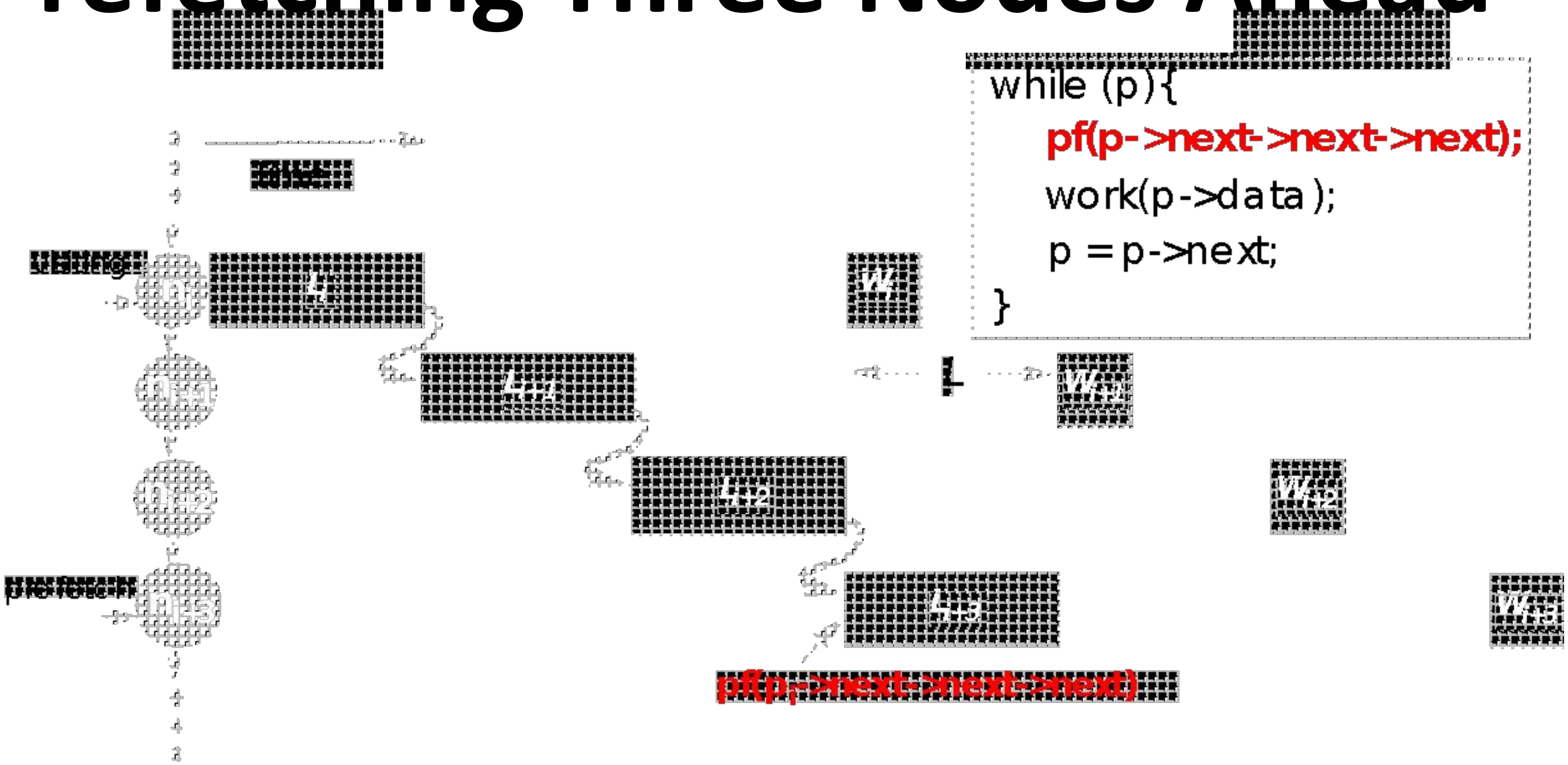


```
while (p){
    pf(p->next);
    work(p->data);
    p =p->next;
}
```

- Computation is overlapped with memory accesses

computation rate = 1/L

# Prefetching Three Nodes Ahead



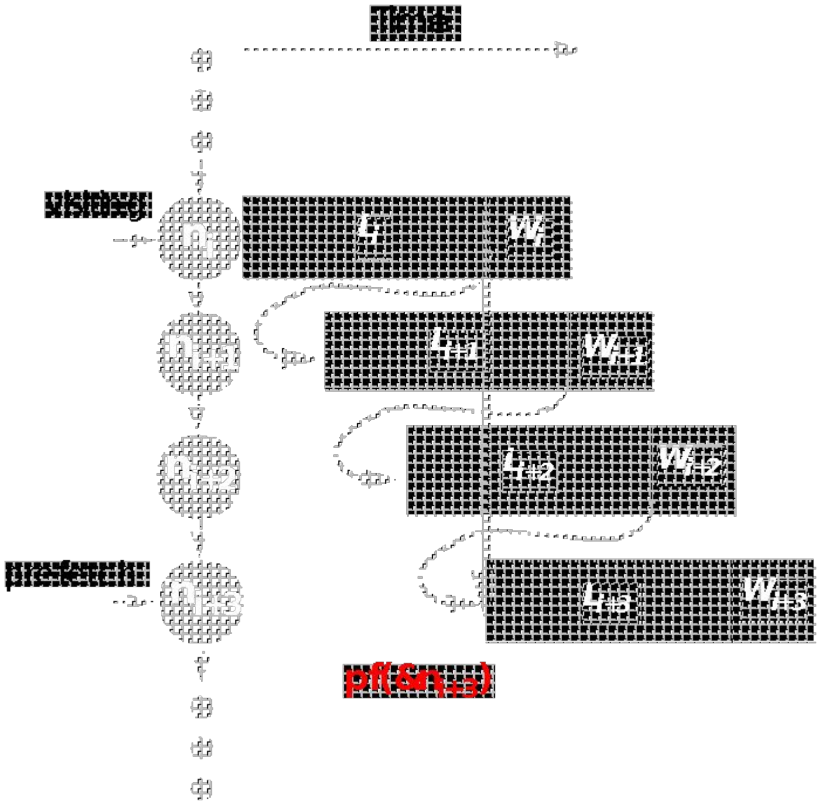
```
while (p){
    pf(p->next->next->next);
    work(p->data);
    p = p->next;
}
```

*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

# Overview

- Challenges in Prefetching Recursive Data Structures
- Three Prefetching Algorithms
  - Greedy Prefetching
  - History-Pointer Prefetching
  - Data-Linearization Prefetching
- Experimental Results
- Conclusions

# Pointer-Chasing Problem

Key:

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

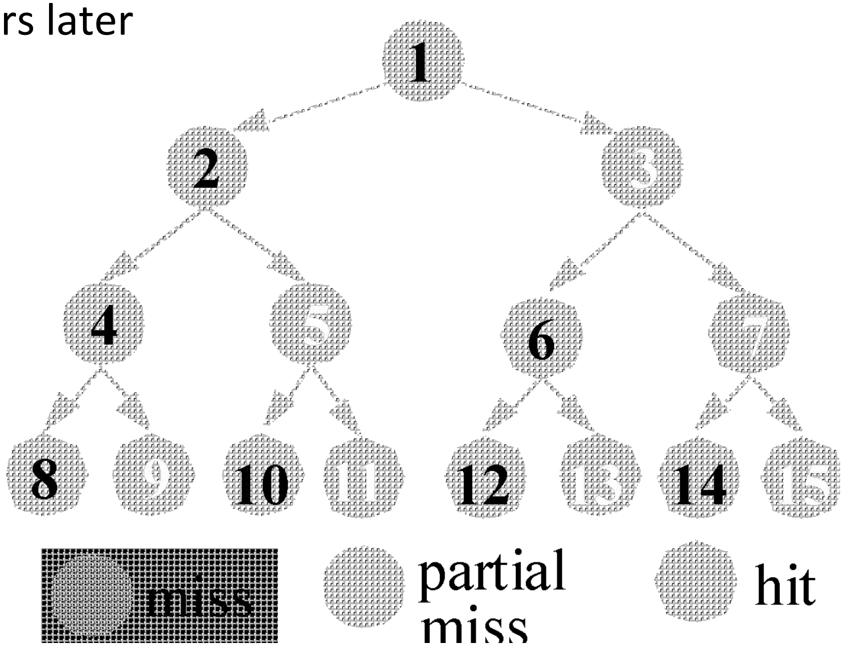
Our proposals:

- 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)
  - History-Pointer 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

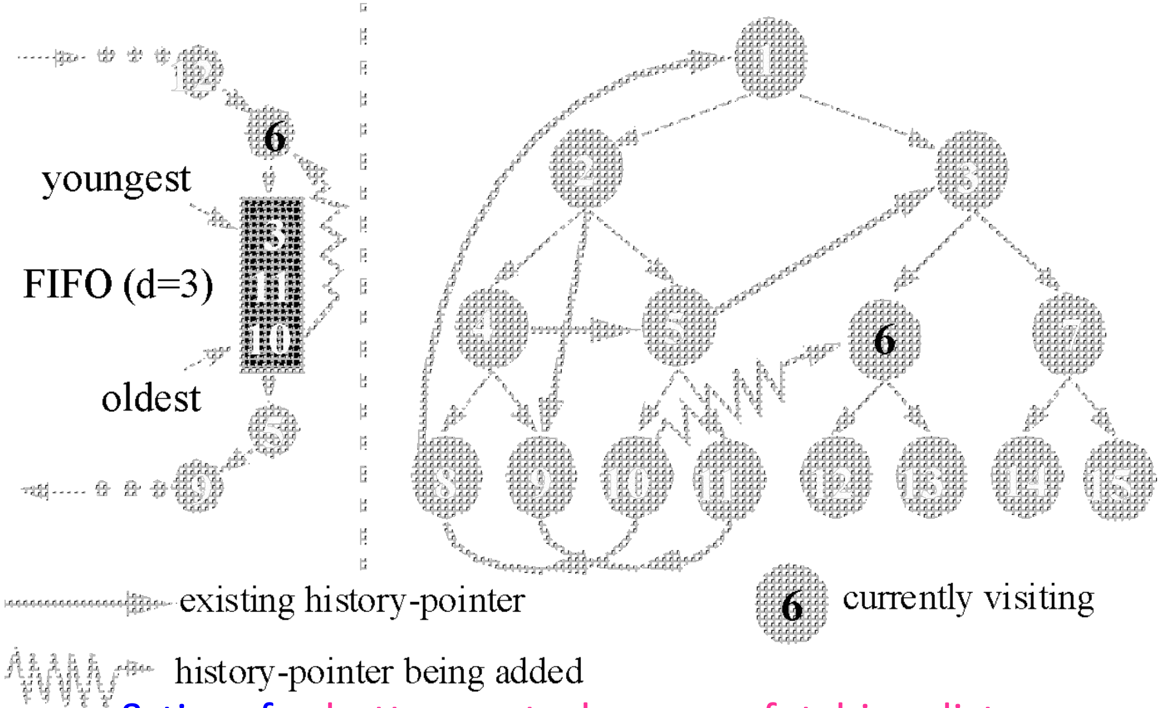
```
preorder(treeNode * t){
  if (t != NULL){
    pf(t->left);
    pf(t->right);
    process(t->data);
    preorder(t->left);
    preorder(t->right);
  }
}
```



- 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



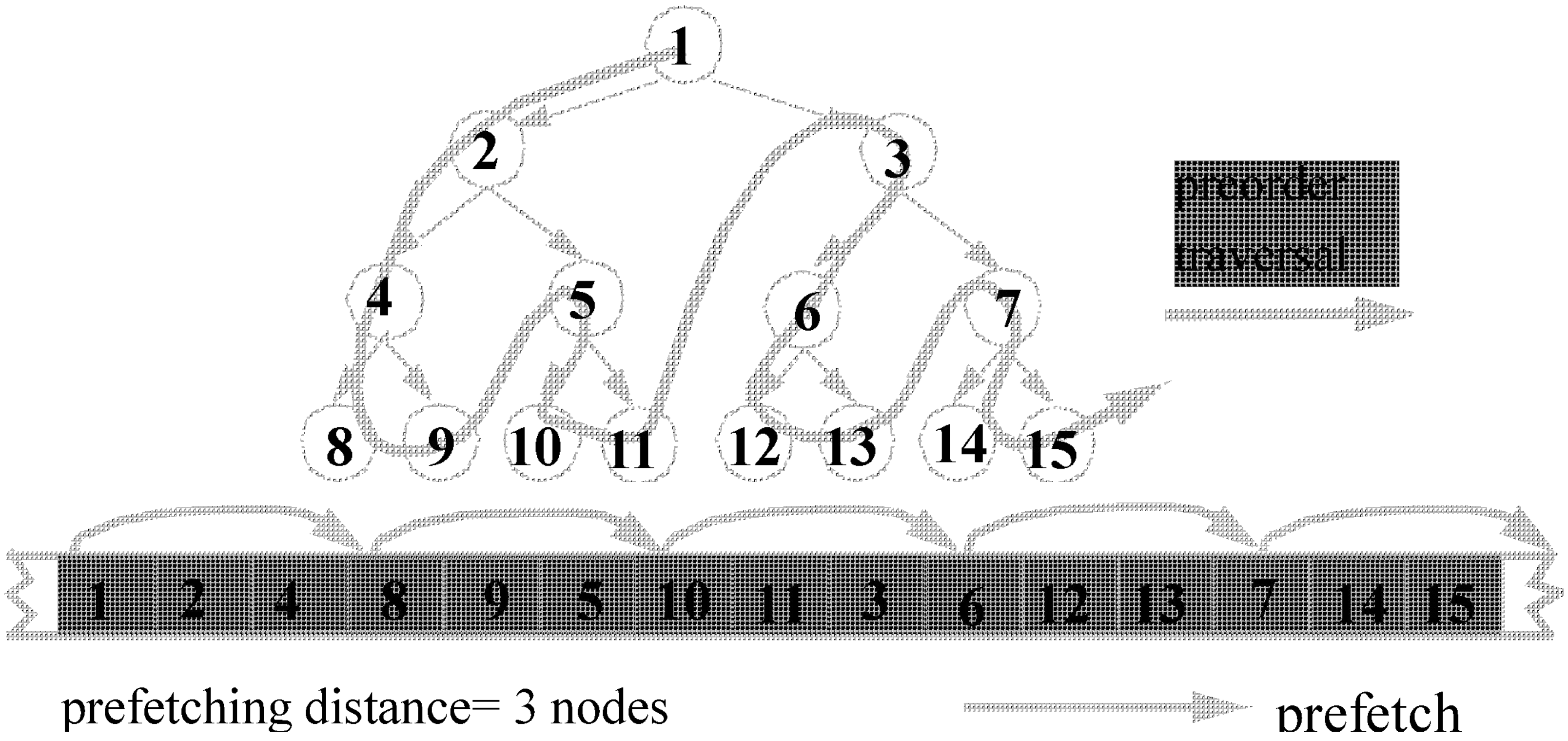
- Trade space & time for better control over prefetching distances

1
2
4
8
9
5
10
11
3
6

preorder

# Data-Linearization Prefetching

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



# Summary of Prefetching Algorithms

| | Greedy | History-Pointer | Data-Linearization |
|---|---|---|---|
| Control over Prefetching Distance | little | more precise | more precise |
| Applicability to Recursive Data Structures | any RDS | 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 |

# Conclusions

- Propose 3 schemes to overcome the pointer-chasing problem:
  - Greedy Prefetching
  - History-Pointer Prefetching
  - Data-Linearization Prefetching

- Automated greedy prefetching in SUIF
  - improves performance significantly for half of Olden
  - memory feedback can further reduce prefetch overhead

- The other 2 schemes can outperform greedy in some situations