Compiler-Based Prefetching for Recursive Data Structures

Chi-Keung Luk and *Todd C. Mowry*

Department of Computer Science

Department of Electrical and Computer Engineering

University of Toronto

The Memory Latency Problem

- Key performance bottleneck in modern computer systems
- Caches are helpful, but not a panacea
- Compiler techniques have been limited to numeric programs

☞ Expand compiler's scope to include non-numeric programs

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:

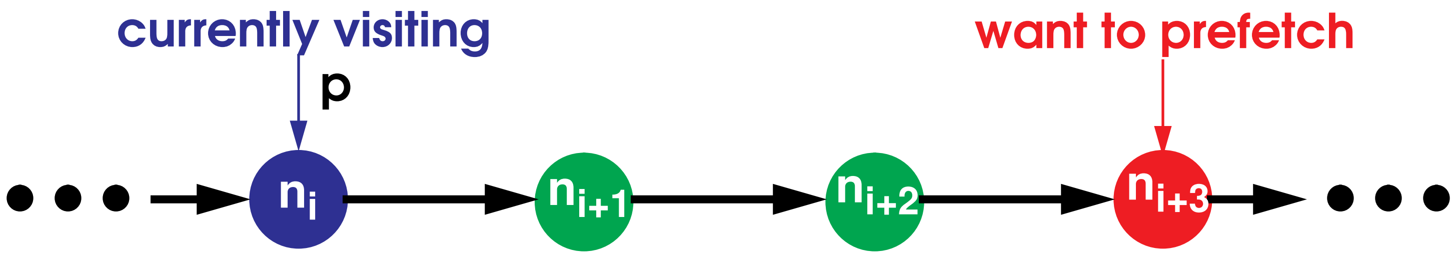
☞ **Automatic Compiler-Based Prefetching for Recursive Data Structures**

Overview

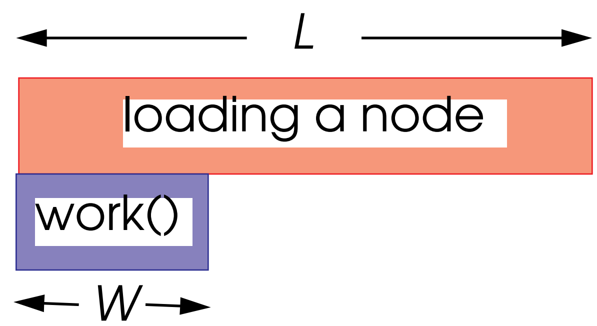
☞ Challenges in Prefetching Recursive Data Structures

- Three Prefetching Algorithms
- Experimental Results
- Conclusions

Scheduling Prefetches for Recursive Data Structures



$p = \&n_0$
while (p){ ← load *p here
    work(p->data);
    p = p->next;
}



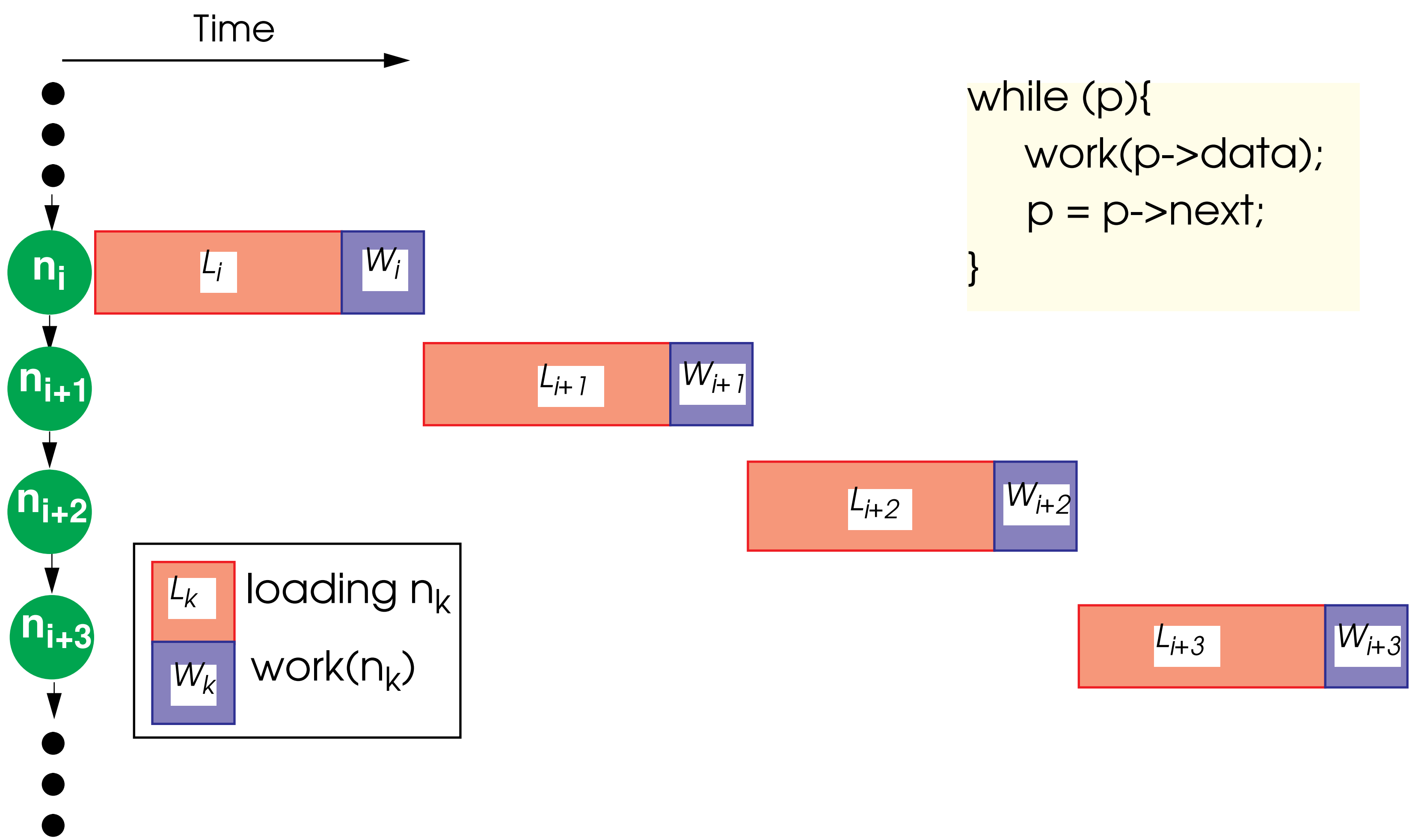
<u>Our Goal:</u> ***fully hide latency***

- **thus achieving fastest possible computation rate of $1/W$**

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

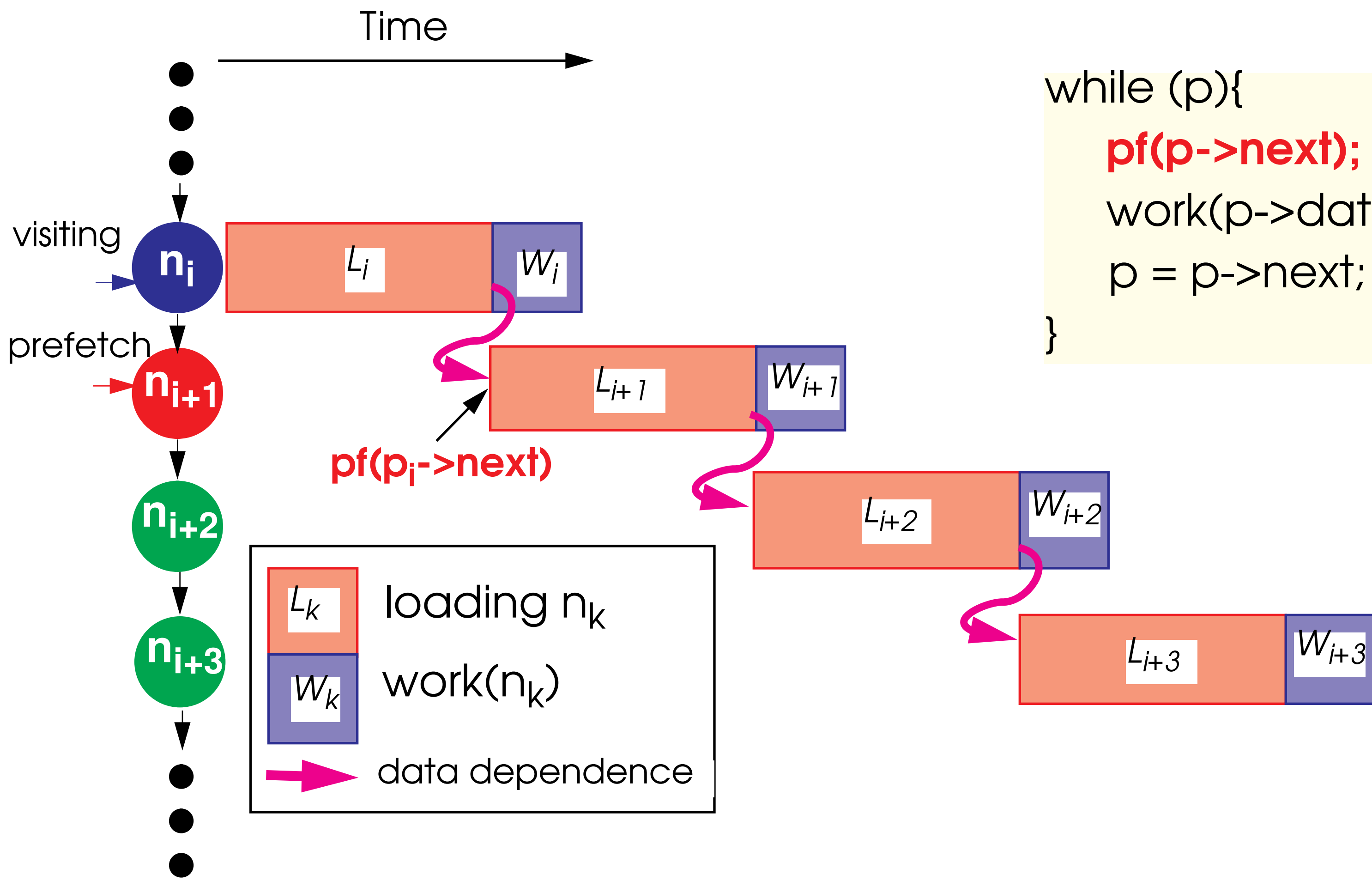
Performance without Prefetching

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



☞ **computation rate = 1/(L+W)**

Prefetching One Node Ahead

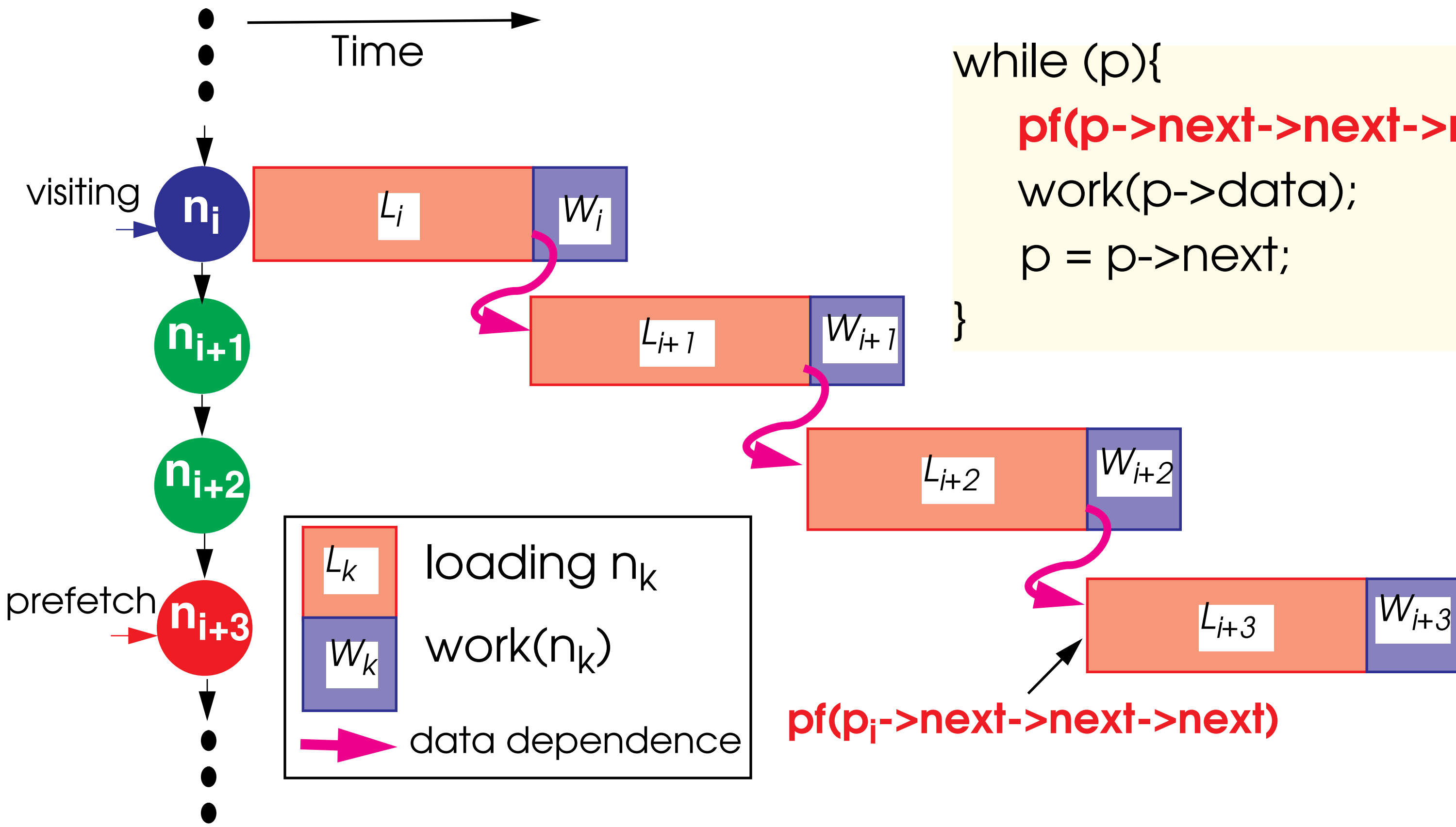



```
while (p){
    pf(p->next);
    work(p->data);
    p = p->next;
}
```

- Computation is overlapped with memory accesses

  ☞ **computation rate = 1/*L***

Prefetching Three Nodes Ahead



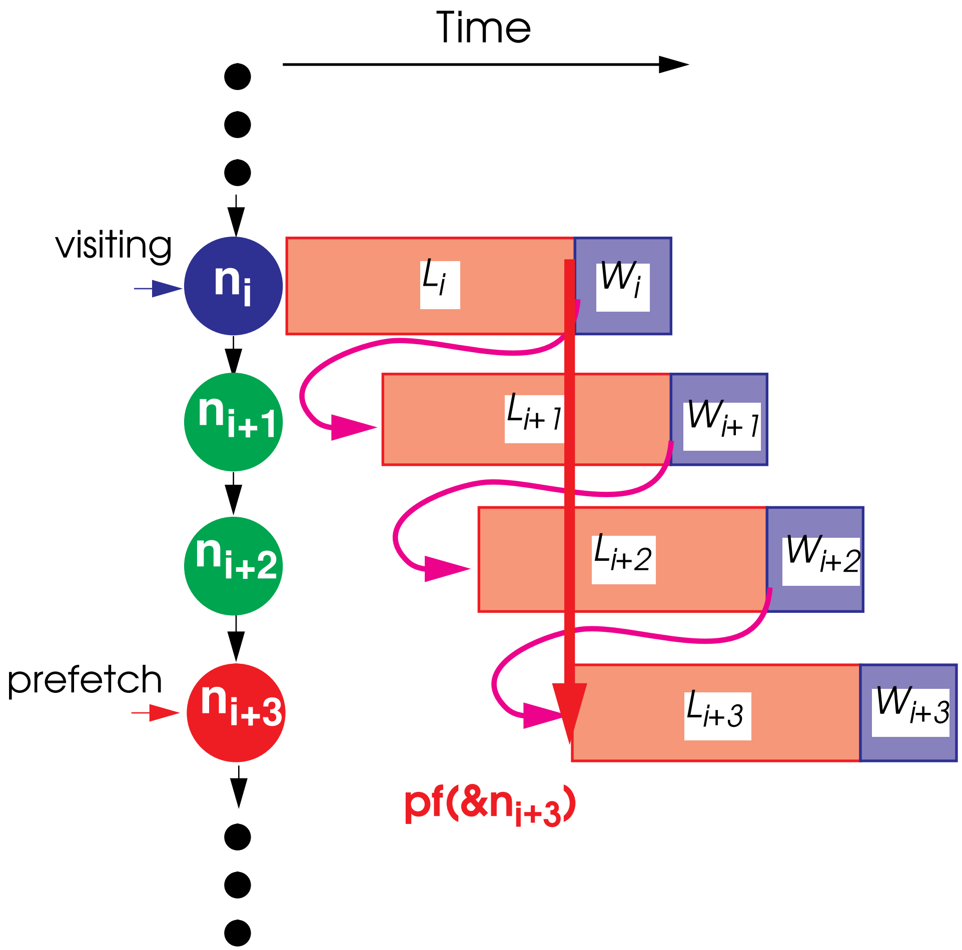

```
while (p){
    pf(p->next->next->next);
    work(p->data);
    p = p->next;
}
```

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

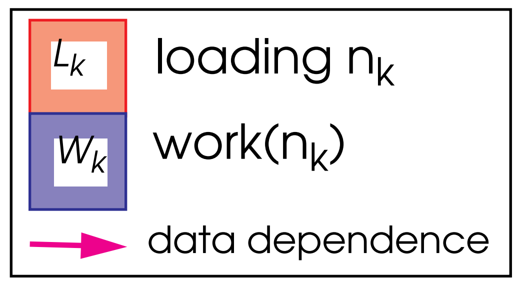
Pointer-Chasing Problem:

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

Our Goal: Fully Hide Latency




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



☞ **achieves the fastest possible computation rate of 1/*W***

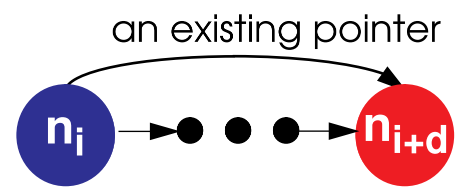
Overcoming the Pointer-Chasing Problem

Key:

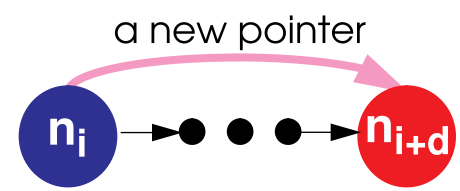
☞ **n_i needs to know $\&n_{i+d}$ without referencing the d-1 intermediate nodes**

Three approaches:

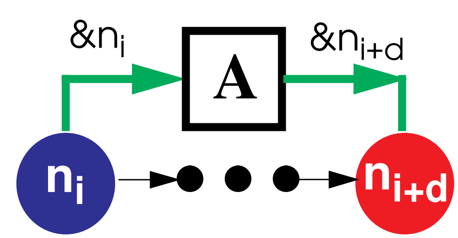
- use *existing* pointer(s) in n_i to approximate $\&n_{i+d}$
  - e.g. **Greedy Prefetching**



- add *new* pointer(s) to n_i to approximate $\&n_{i+d}$
  - e.g. **History-Pointer Prefetching**



- compute $\&n_{i+d}$ *directly* from $\&n_i$ (no ptr. deref.)
  - e.g. **Data-Linearization Prefetching**



A=Address generating function

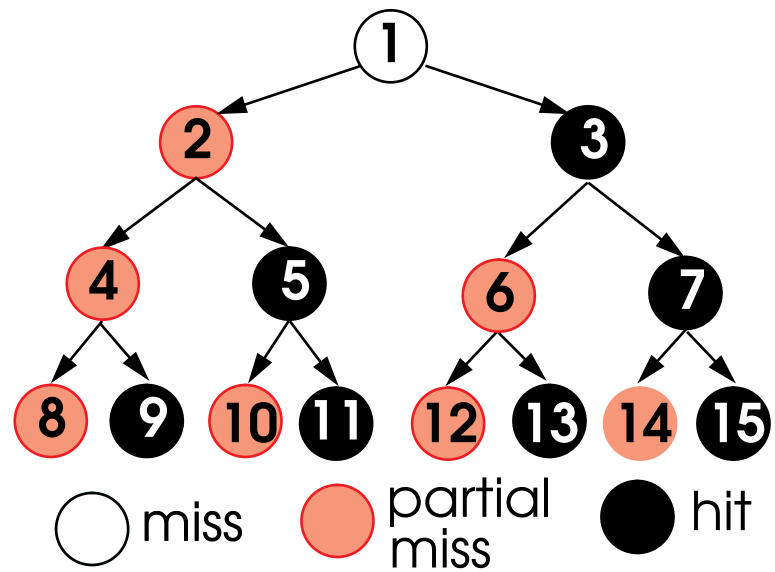
Overview

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

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

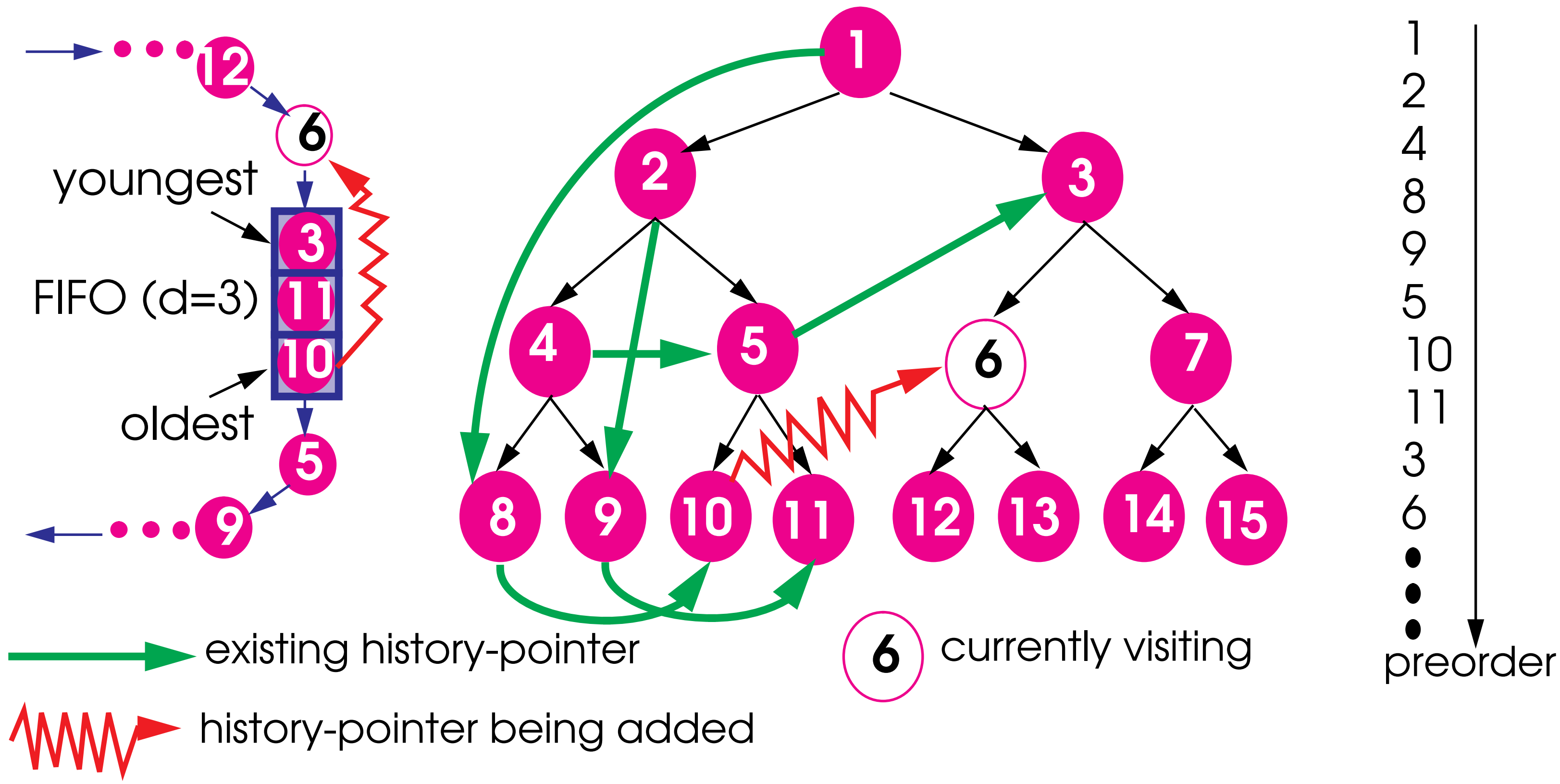
```
preorder(treeNode * t){
   if (t != NULL){
        pf(t->left);
        pf(t->right);
        process(t->data);
        preorder(t->left);
        preorder(t->right);
   }
}
```



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

History-Pointer Prefetching

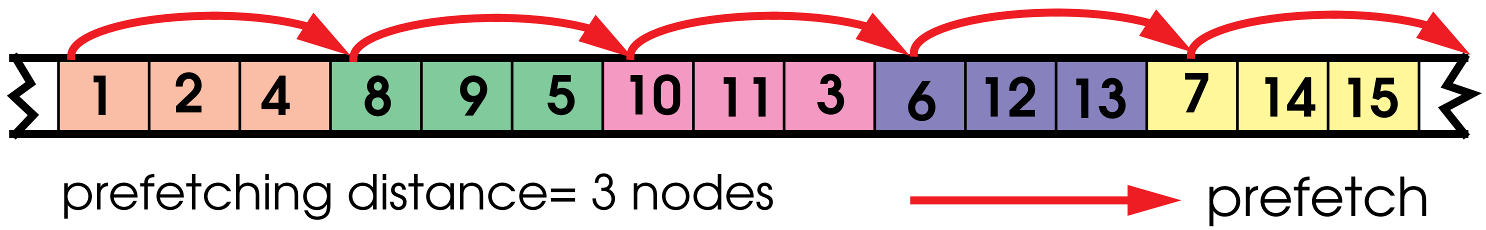
- Add new pointer(s) to each node
  - history-pointers are obtained from some recent traversal

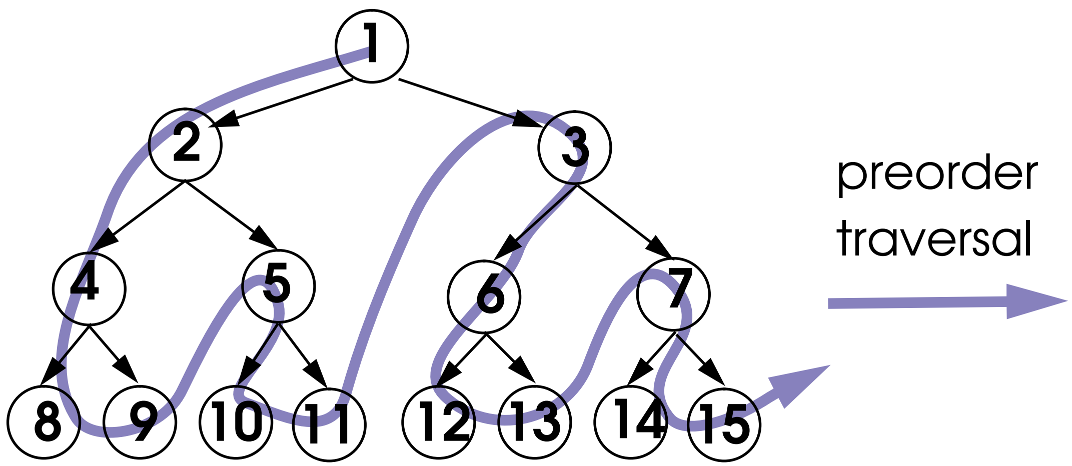


- Trade space & time for better control over prefetching distances

Data-Linearization Prefetching

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





Summary of Prefetching Algorithms

	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 straight-forward	more difficult	more difficult

Greedy prefetching is the most widely applicable algorithm:

☞ **fully implemented in SUIF**

Overview

✓ Challenges in Prefetching Recursive Data Structures

✓ Three Prefetching Algorithms

☞ Experimental Results

- Conclusions

Experimental Framework

Benchmarks

- Olden benchmark suite
  - 10 pointer-intensive programs
  - covers a wide range of recursive data structures

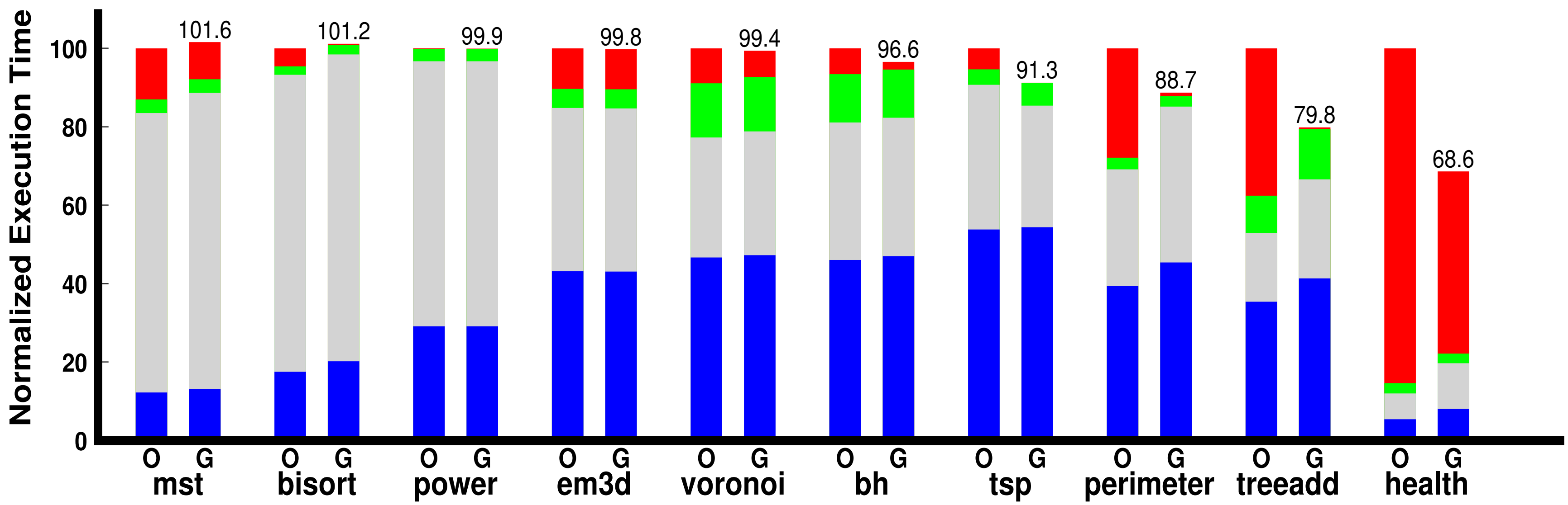
Simulation Model

- Detailed, cycle-by-cycle simulations
  - MIPS R10000-like dynamically-scheduled superscalar

Compiler

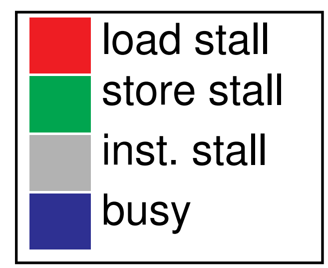
- Implemented in the SUIF compiler
- Generates fully functional, optimized MIPS binaries

Performance of Compiler-Inserted Greedy Prefetching



O = Original

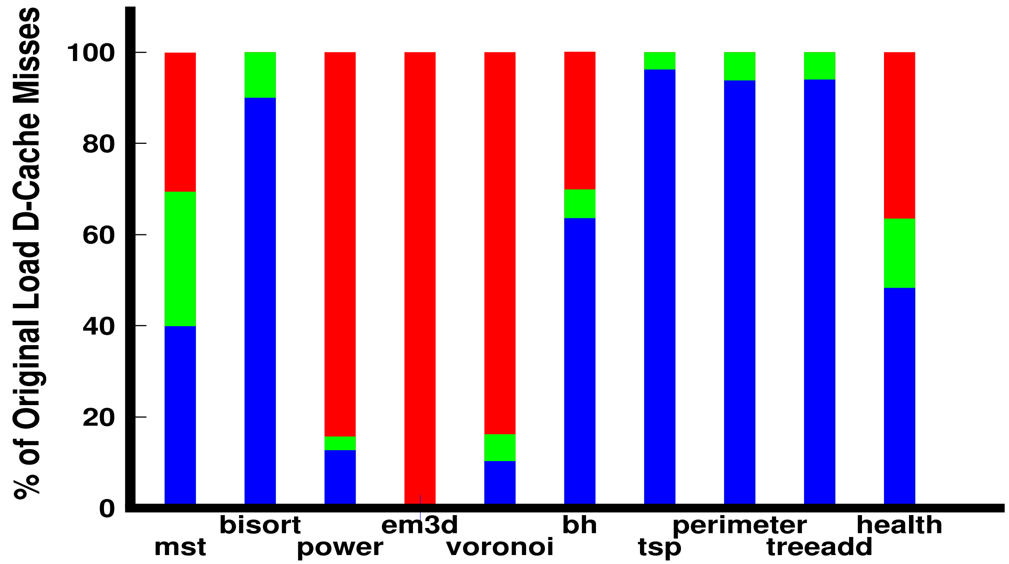
G = Compiler-Inserted Greedy Prefetching



- Eliminates much of the stall time in programs with large load stall penalties

  ☞ **half achieve speedups of 4% to 45%**

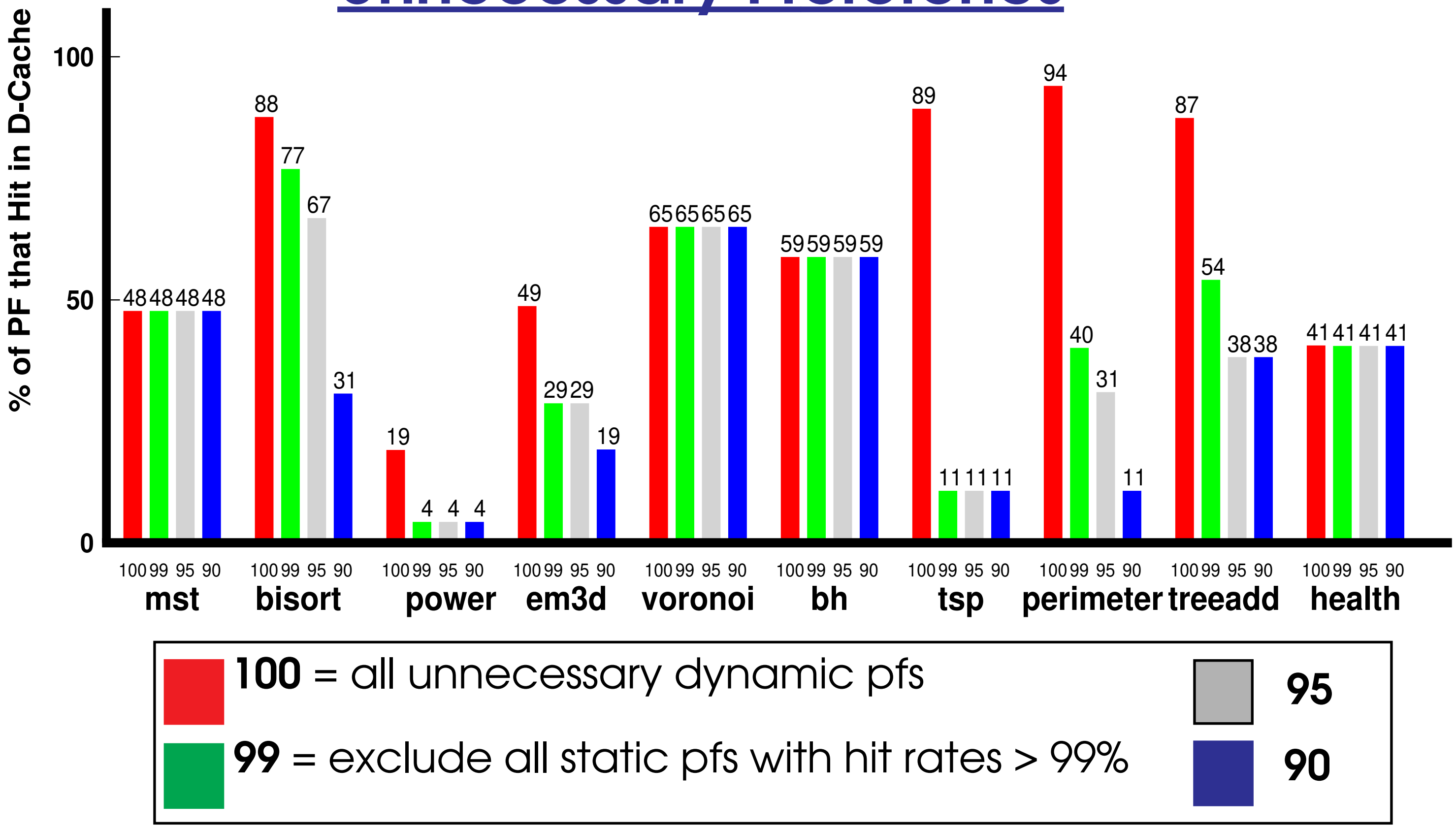
Coverage Factor



nopf_miss = original D-cache misses that are not prefetched
pf_miss = original D-cache misses that are prefetched but remain misses
pf_hit = original D-cache misses that are prefetched and then hit in the D-cache

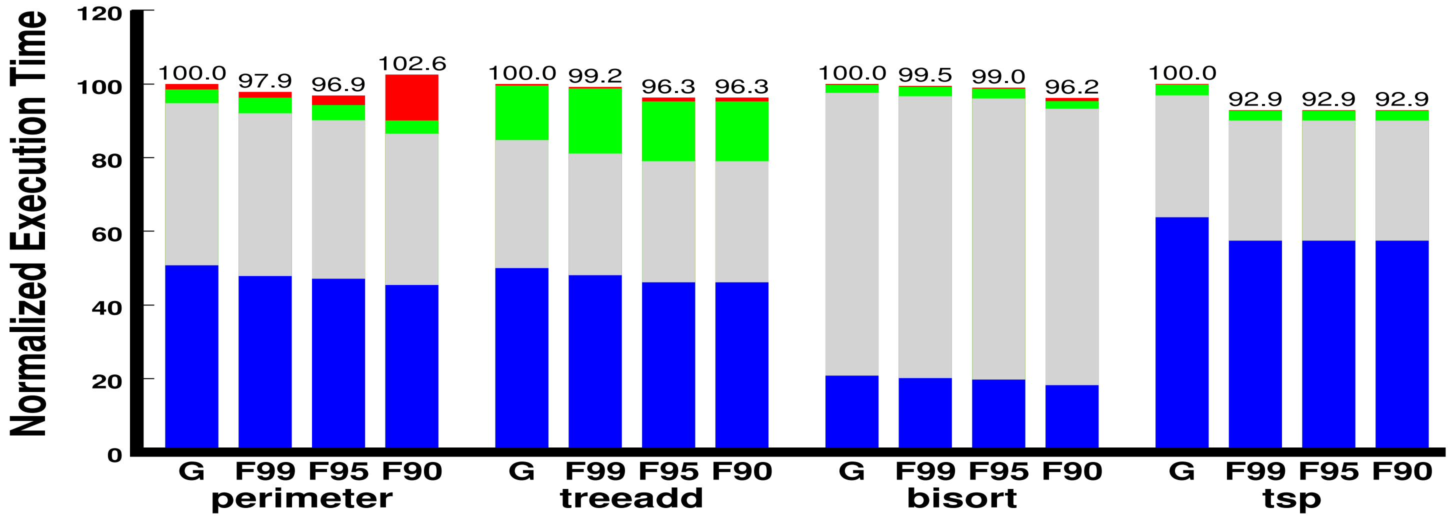
- coverage factor = pf_hit + pf_miss
- 7 out of 10 have coverage factors > 60%
  - em3d, power, voronoi have many array or scalar load misses
- small pf_miss fractions => effective prefetch scheduling

Unnecessary Prefetches



- % dynamic pfs that are unnecessary because the data is in the D-cache
- 4 have >80% unnecessary prefetches
- Could reduce overhead by eliminating static pfs that have high hit rates

Reducing Overhead Through Memory Feedback



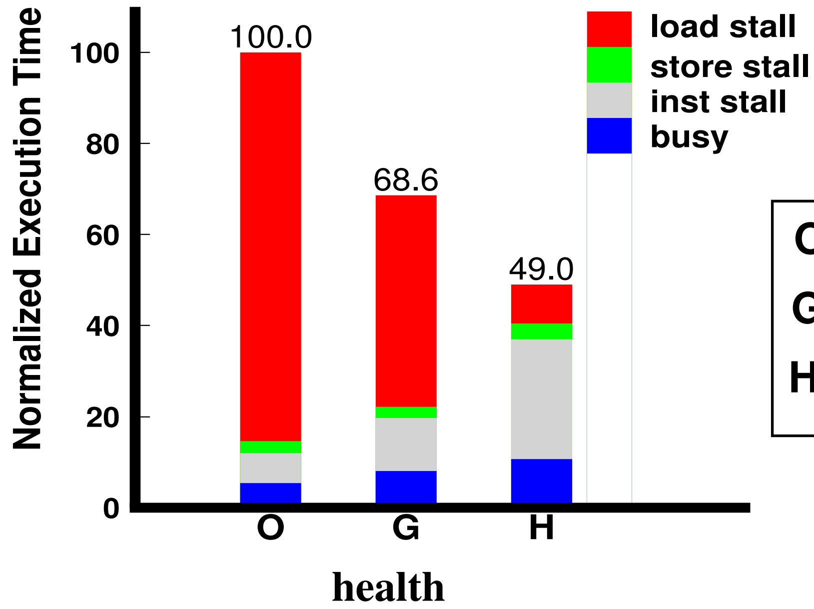
G = greedy prefetching
Fxx = greedy prefetching where static pfs with hit rate > xx% are eliminated

load stall
store stall
inst. stall
busy

- Eliminating static pfs with hit rate >95% speeds them up by 1-8%
- However, eliminating useful prefetches can hurt performance

☞ **Memory feedback can potentially improve performance**

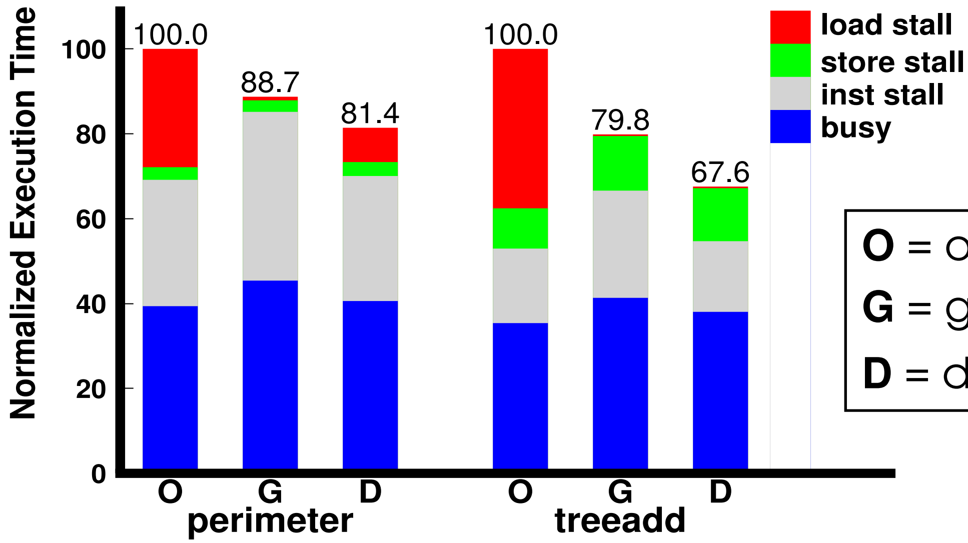
Performance of History-Pointer Prefetching



O = original
G = greedy prefetching
H = history-pointer prefetching

- Applicable because a list structure does not change over time
- 40% speedup over greedy prefetching through:
    - better miss coverage (64% -> 100%)
    - fewer unnecessary prefetches (41% -> 29%)
- Improved accuracy outweighs increased overhead in this case

Performance of Data-Linearization Prefetching



O = original
G = greedy prefetching
D = data-linearization prefetching

- Creation order equals major traversal order in treeadd & perimeter
  - hence data linearization is done without data restructuring
- 9% and 18% speedups over greedy prefetching through:
  - fewer unnecessary prefetches:
    - 94%->78% in perimeter, 87%->81% in treeadd
  - while maintaining good coverage factors:
    - 100%->80% in perimeter, 100%->93% in treeadd

Other Results

- **Architectural Support Needed:**
  - at least 2 memory units
  - non-excepting prefetches

- **Quantitative Comparison with Related Work:**
  - Greedy prefetching outperforms "SPAID" by:
    - prefetching references that are likely to miss
    - prefetching further ahead

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**

☞ ***Encouraging results suggest that compiler-based prefetching for non-numeric codes is a promising area for future research***