# Lecture 22

# Prefetching Recursive Data Structures

Material from: C.-K. Luk and T. C. Mowry. "Compiler-Based Prefetching for Recursive Data Structures." In Proceedings of ASPLOS-VII, Oct. 1996, pp. 222-233.



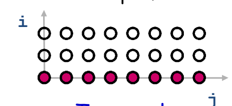
## Recall: Loop Splitting for Prefetching Arrays

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

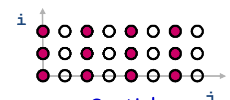
| Locality Type | Predicate | Loop Transformation |
|---|---|---|
| None | True | None |
| Temporal | i = 0 | Peel loop i |
| Spatial | (i mod L) = 0 | Unroll loop i by L |

(L elements/cache line)

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




Temporal




Spatial



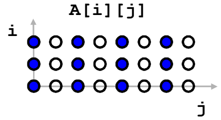
## Recall: Example Code with Prefetching Arrays

Original Code

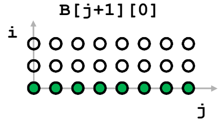
```
for (i = 0; i < 3; i++)
  for (j = 0; j < 100; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```

○ 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];
  }
}
```



## 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:

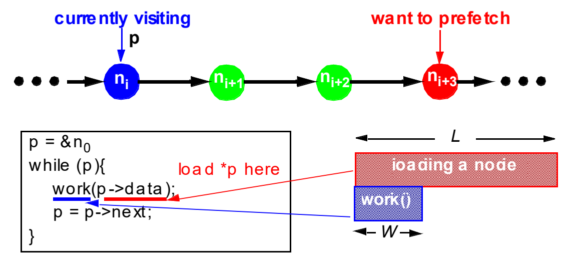
- Automatic Compiler-Based Prefetching for Recursive Data Structures

# Overview

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



# Scheduling Prefetches for Recursive Data Structures



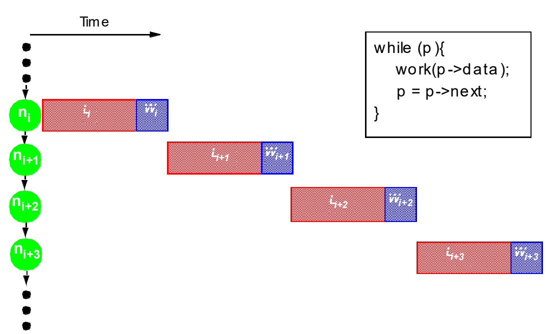

Our Goal: *fully hide latency*

- thus achieving fastest possible computation rate of 1/W

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



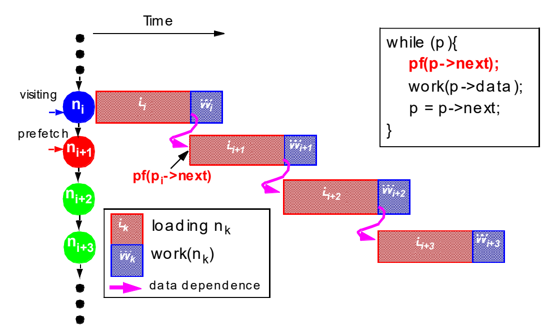
# Performance without Prefetching




computation rate = 1 / (L+W)



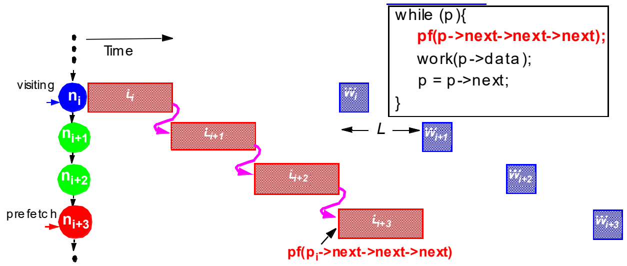
# Prefetching One Node Ahead




- Computation is overlapped with memory accesses

computation rate = 1/L

## Prefetching Three Nodes Ahead



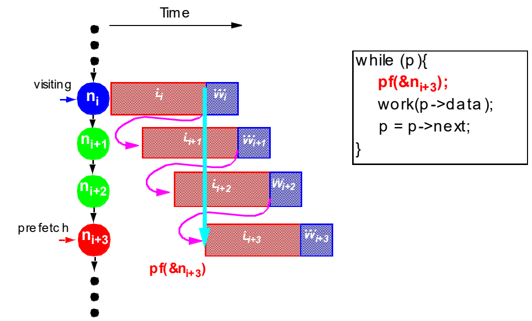

*computation rate does not improve (still = 1/L)!*

Pointer-Chasing Problem:

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



## Our Goal: Fully Hide Latency




- 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



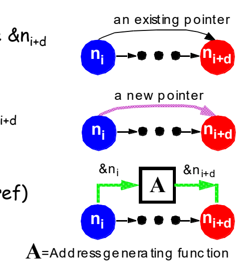
## Overcoming the Pointer-Chasing Problem

Key:

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

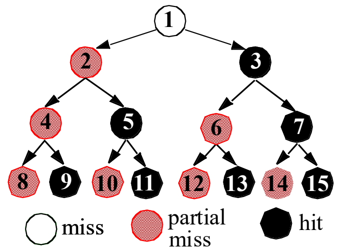
Our proposals:

- 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



- 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);
  }
}
```

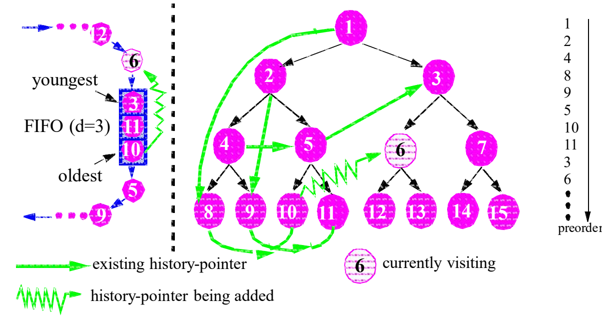



- Reasonably effective in practice
- However, little control over the prefetching distance



## History-Pointer Prefetching

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

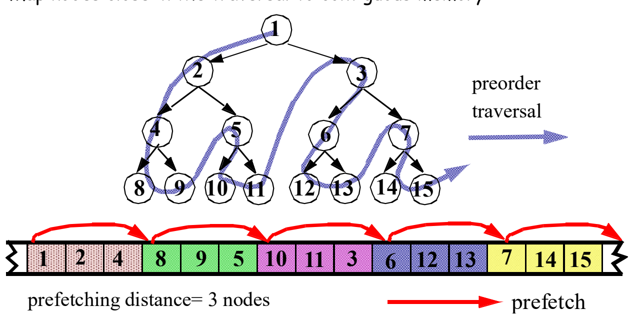



- Trade space & time for better control over prefetching distances



## 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 straightforward | more difficult | more difficulty |

- 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

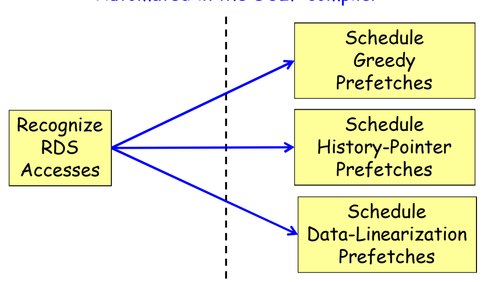
Simulation Model
- Detailed, cycle-by-cycle simulations
- MIPS R10000-like dynamically-scheduled superscalar

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



## Implementation of Prefetching Algorithms

Automated in the SUIF compiler

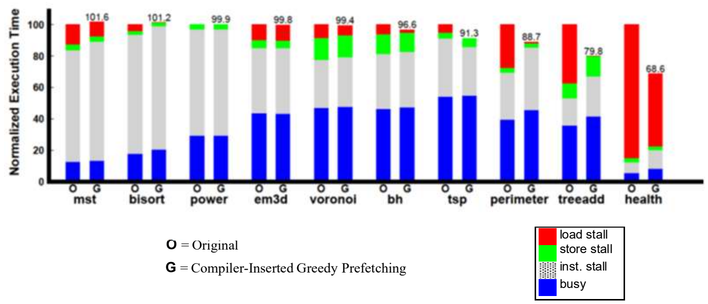



- identify RDS types
- find recurrent pointer updates in loops and recursive procedures

- insert prefetches at the earliest possible places
- minimize prefetching overhead

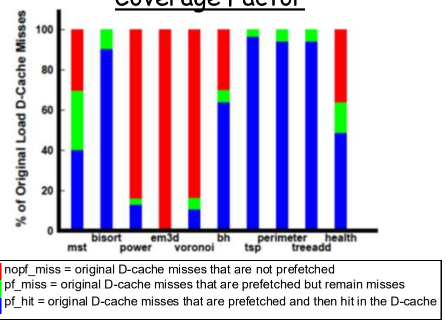


## Performance of Compiler-Inserted Greedy Prefetching




- Eliminates much of the stall time in programs with large load stall penalties
  - half achieve speedups of 4% to 45%

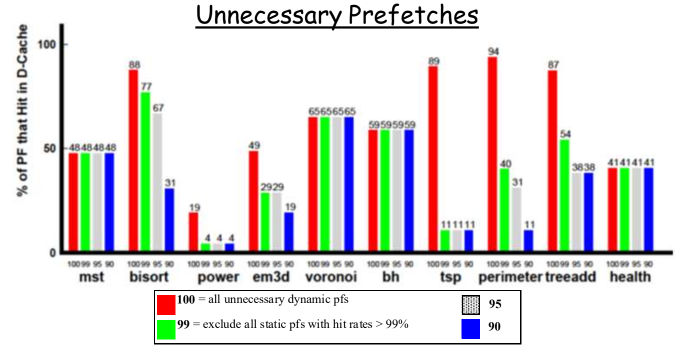
## Coverage Factor




- 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



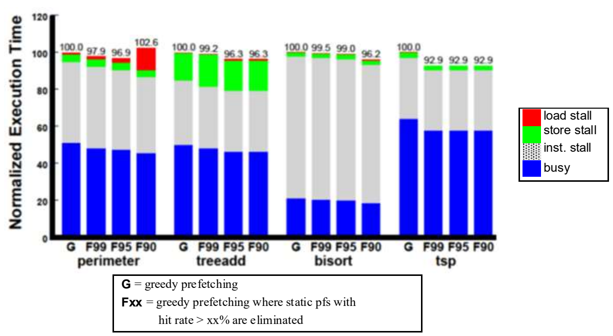
## Unnecessary Prefetches




- % 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



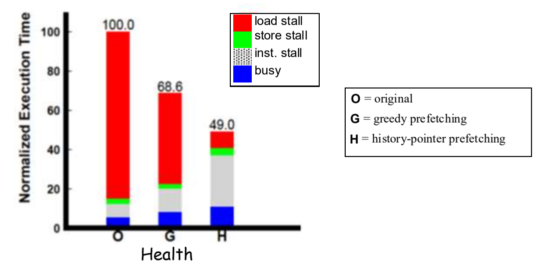
## Reducing Overhead Through Memory Feedback




- 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

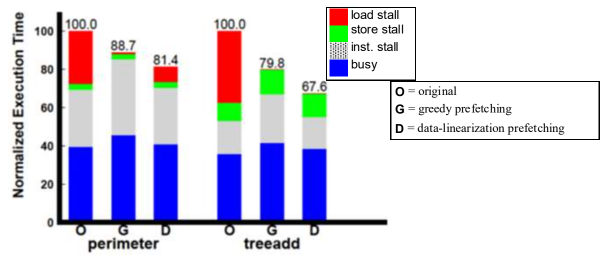


## Performance of History-Pointer Prefetching




- 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




- 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



## Conclusions

- Three 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



## Monday's Class

- Register Allocation - Coalescing