Lecture 27

Compiler Algorithms for Prefetching Data

I. Prefetching for Arrays

II. Prefetching for Recursive Data Structures

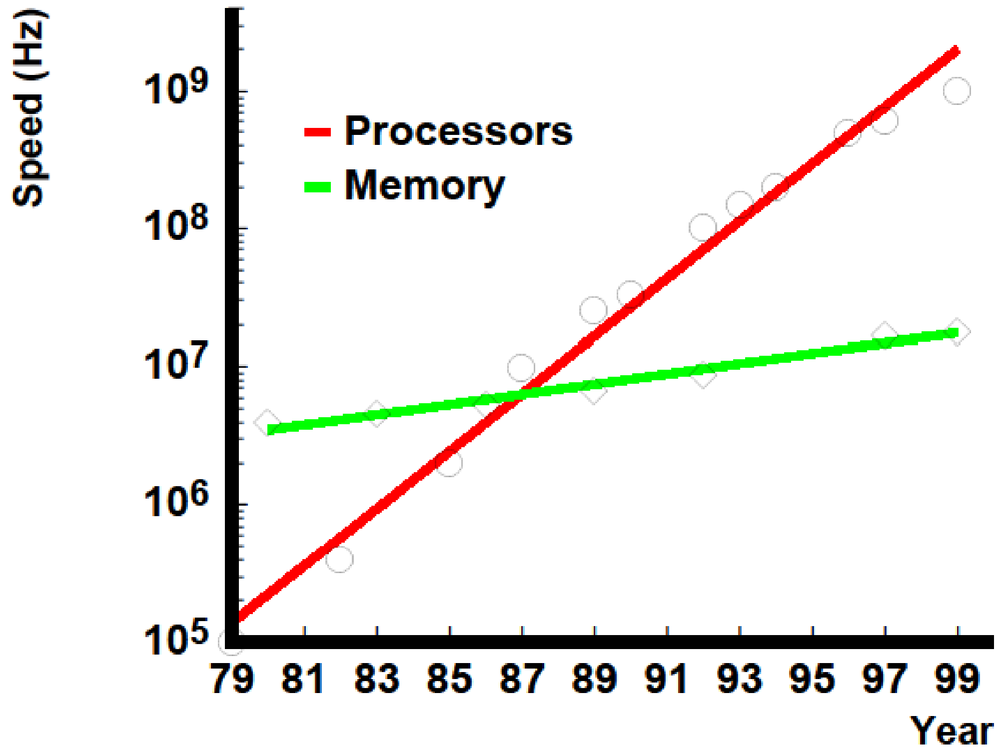
Reading: ALSU 11.11.4

<u>Advanced readings (optional):</u>

T.C. Mowry, M. S. Lam and A. Gupta. "Design and Evaluation of a Compiler Algorithm for Prefetching." In Proceedings of ASPLOS-V, Oct. 1992, pp. 62-73.

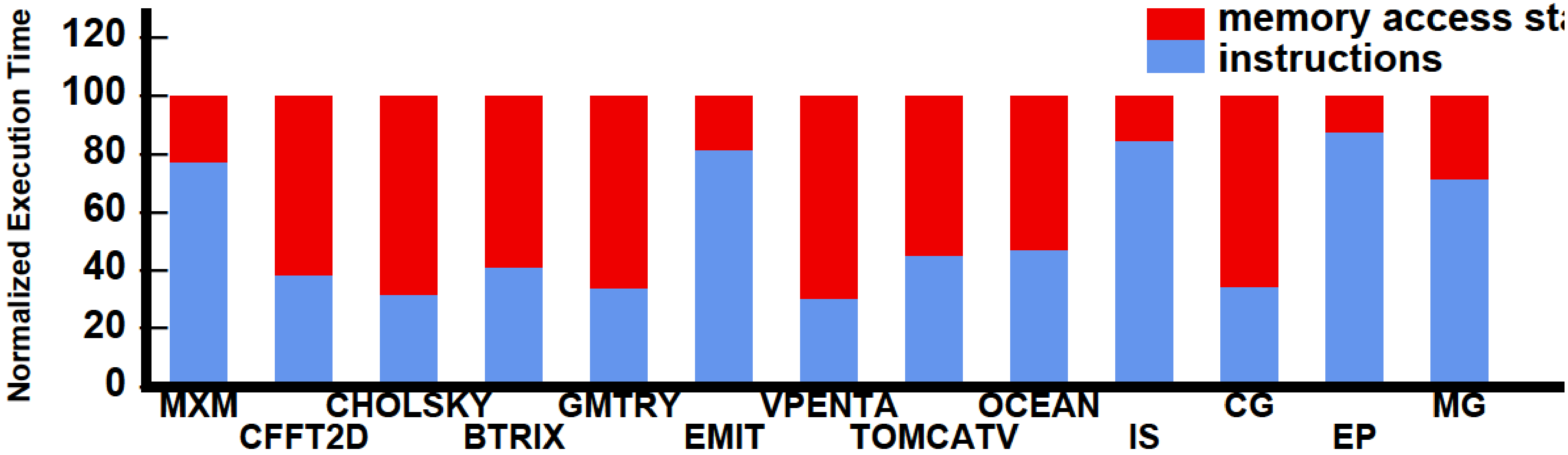
C.-K. Luk and T. C. Mowry. "Compiler-Based Prefetching for Recursive Data Structures." In Proceedings of ASPLOS-VII, Oct. 1996, pp. 222-233.

The Memory Latency Problem



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

Uniprocessor Cache Performance on Scientific Code



- Applications from SPEC, SPLASH, and NAS Parallel.
- Memory subsystem typical of MIPS R4000 (100 MHz):
  - 8K / 256K direct-mapped caches, 32 byte lines
  - miss penalties: 12 / 75 cycles
- 8 of 13 spend > 50% of time stalled for memory

Prefetching for Arrays: Overview

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

Coping with Memory Latency

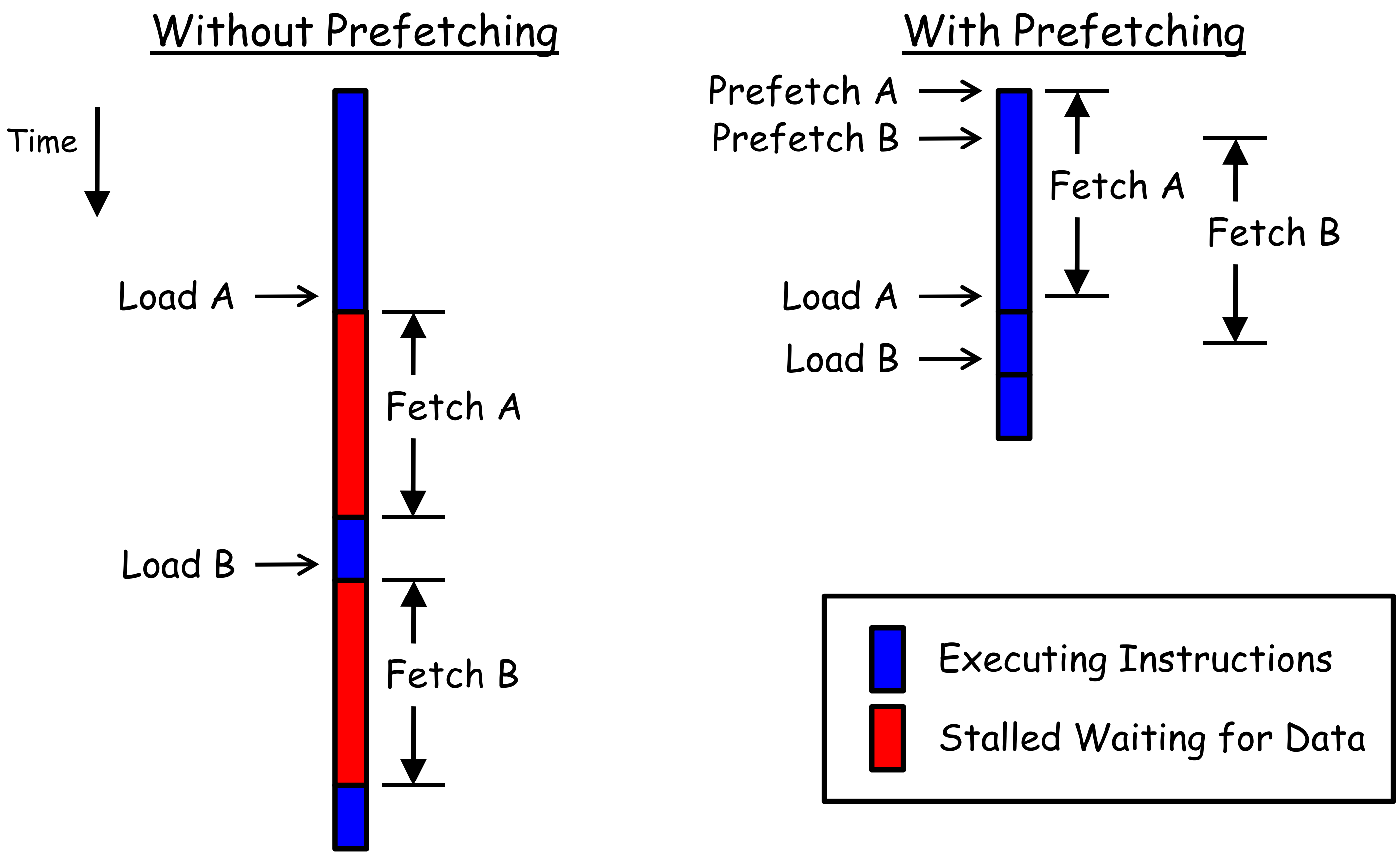
Reduce Latency:

- 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 Research 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

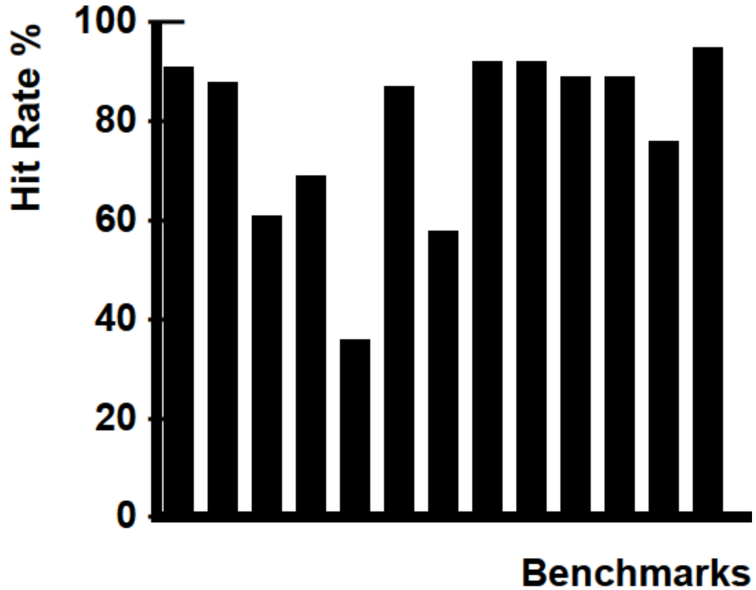
Scheduling: when/how to schedule prefetches

- 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

Compilier Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

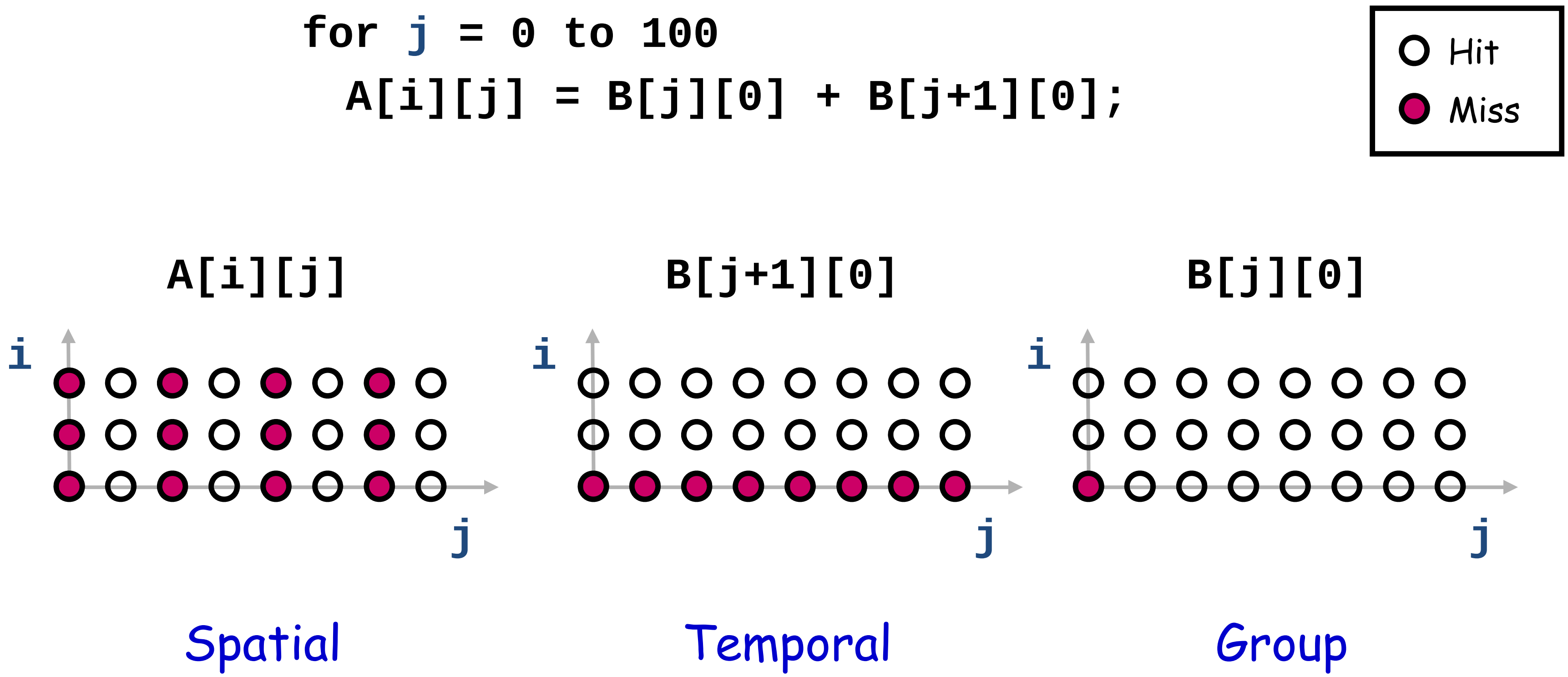
- Loop Splitting
- Software Pipelining

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 $\cap$ localized iteration space $\Rightarrow$ locality

Data Locality Example

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



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 of $\vec{\imath}1$ and $\vec{\imath}2$, we say that reuse occurs along direction vector $\vec{r}$ 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:

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

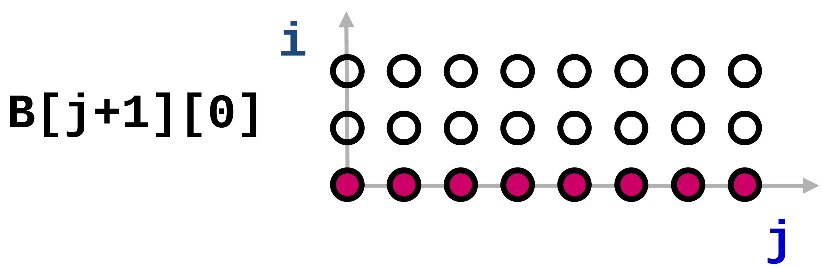
$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- 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).

Localized Iteration Space

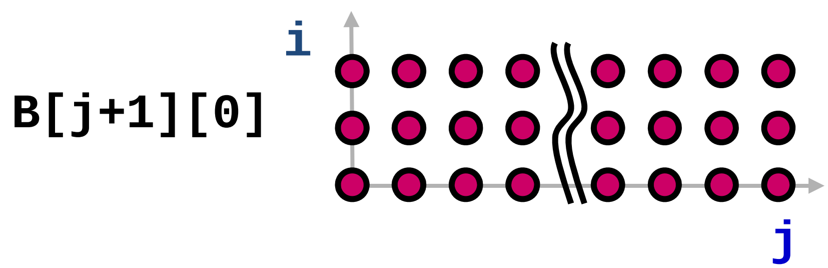
- Given finite cache, when does reuse result in locality?

```
for i = 0 to 2
  for j = 0 to 8
    A[i][j] = B[j][0] + B[j+1][0];
```



Localized: both i and j loops
(i.e. span{(1,0),(0,1)})

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



Localized: j loop only
(i.e. span{(0,1)})

- Localized if accesses less data than *effective cache size*

Computing Locality

- Reuse Vector Space $\cap$ Localized Vector Space $\Rightarrow$ Locality Vector Space
- Example:

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

- If both loops are localized:
  - span{(1,0)} $\cap$ span{(1,0),(0,1)} $\Rightarrow$ span{(1,0)}
  - i.e. temporal reuse *does* result in temporal locality
- If only the innermost loop is localized:
  - span{(1,0)} $\cap$ span{(0,1)} $\Rightarrow$ span{}
  - i.e. no temporal locality

Prefetch Predicate

Locality Type	Miss Instance	Predicate
None	Every Iteration	True
Temporal	First Iteration	**i** = 0
Spatial	Every l iterations (l = cache line size)	(**i** mod 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]	$\begin{bmatrix} \mathtt{i} \\ \mathtt{j} \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$	(**j** mod 2) = 0
B[j+1][0]	$\begin{bmatrix} \mathtt{i} \\ \mathtt{j} \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$	**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

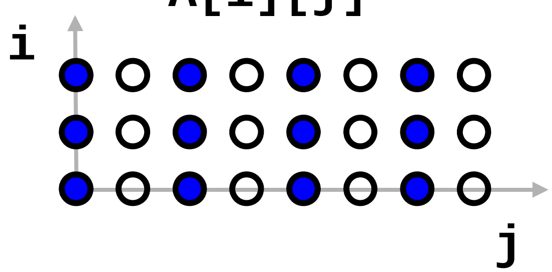
Original Code

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

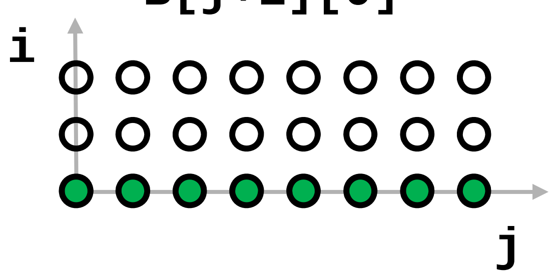
○ Cache Hit

● ● 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];
  }
}
```

Experimental Framework (Uniprocessor)

Architectural Extensions:

- Prefetching support:
  - lockup-free caches
  - 16-entry prefetch issue buffer
  - prefetch directly into both levels of cache
- Contention:
  - memory pipelining rate = 1 access every 20 cycles
  - primary cache tag fill = 4 cycles
- Misses get priority over prefetches

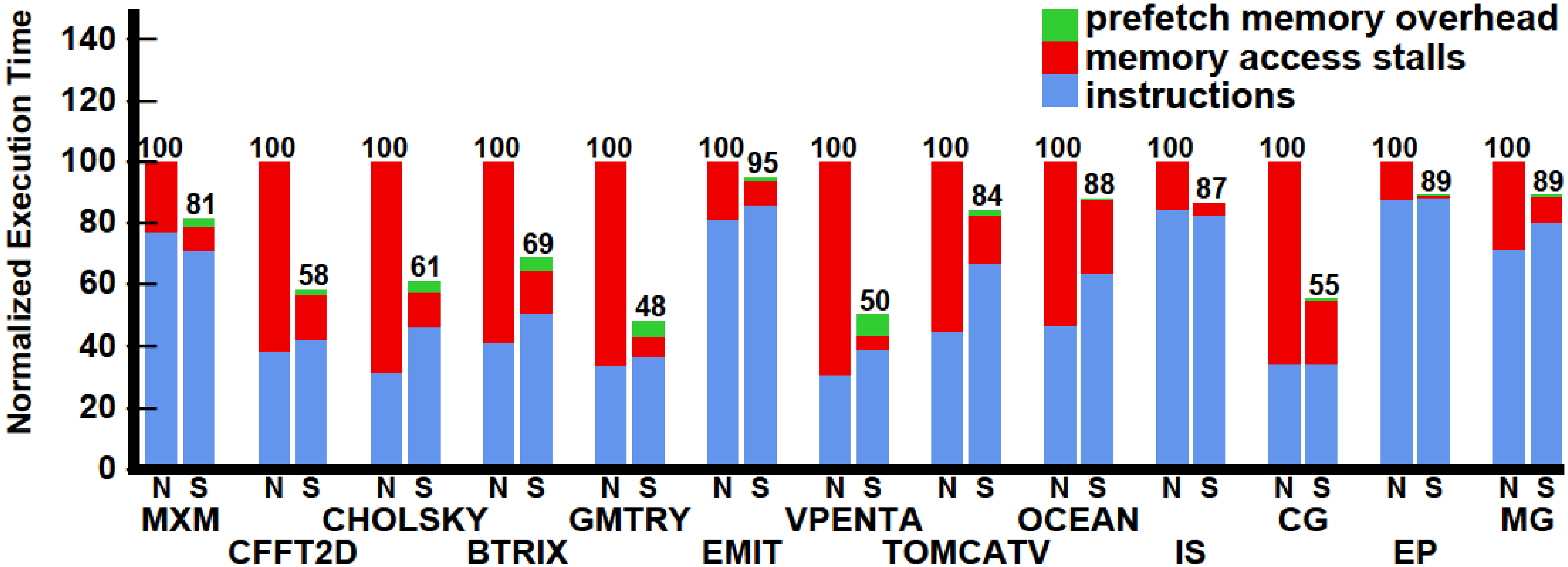
Simulator:

- detailed cache simulator driven by *pixified* object code.

Experimental Results (Dense Matrix Uniprocessor)

- Performance of Prefetching Algorithm
  - Locality Analysis
  - Software Pipelining
- Interaction with Locality Optimizer

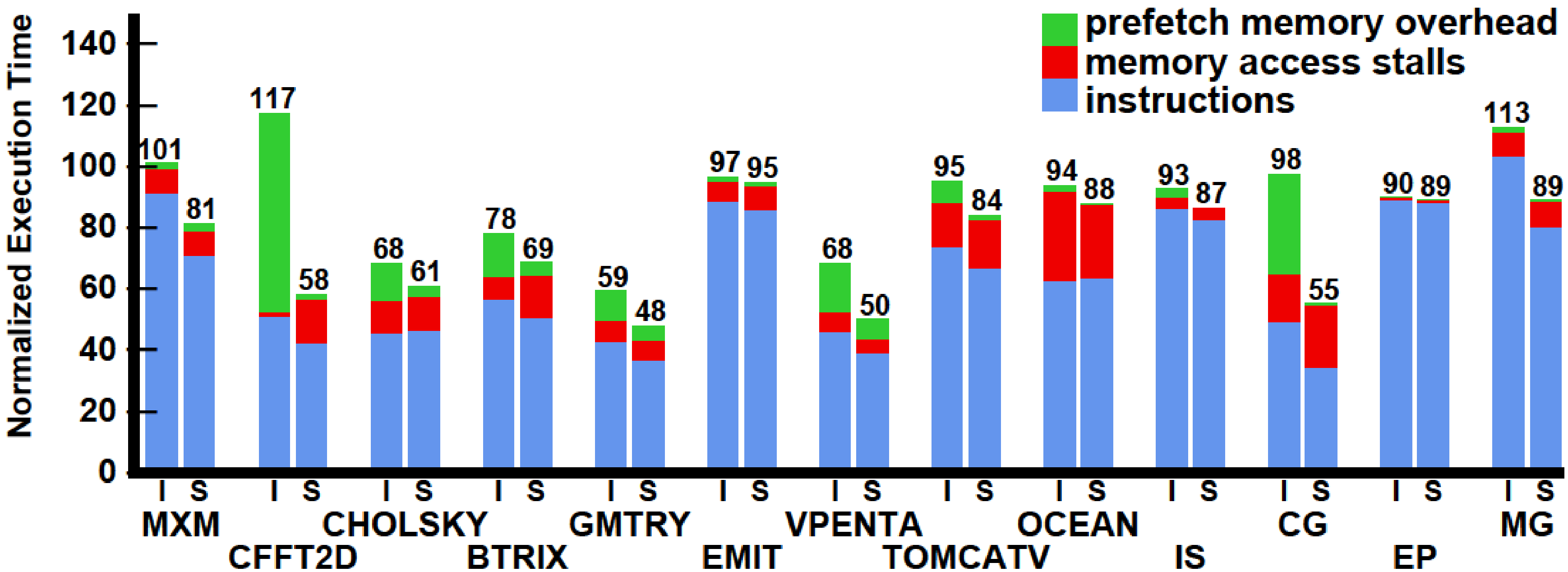
Performance of Prefetching Algorithm



(**N** = No Prefetching, **S** = Selective Prefetching)

- memory stalls reduced by 50% to 90%
- instruction and memory overheads typically low
- 6 of 13 have speedups over 45%

Effectiveness of Locality Analysis

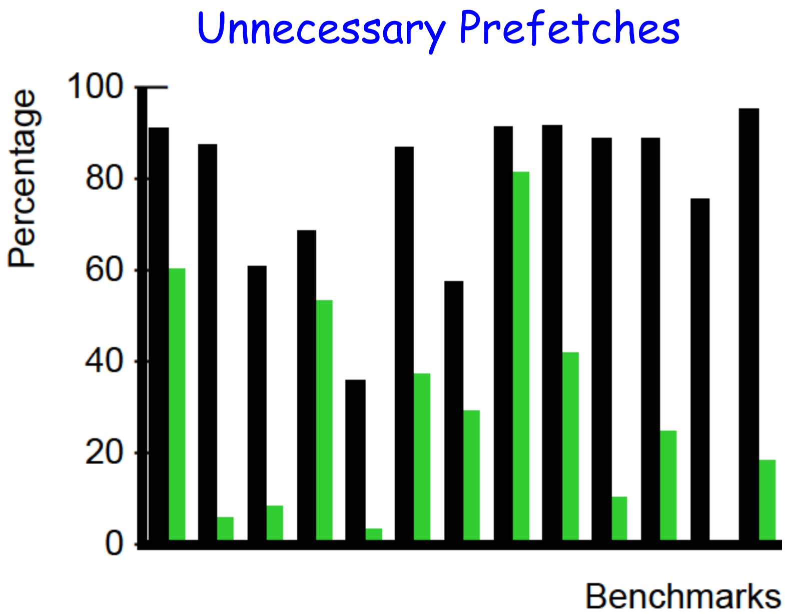


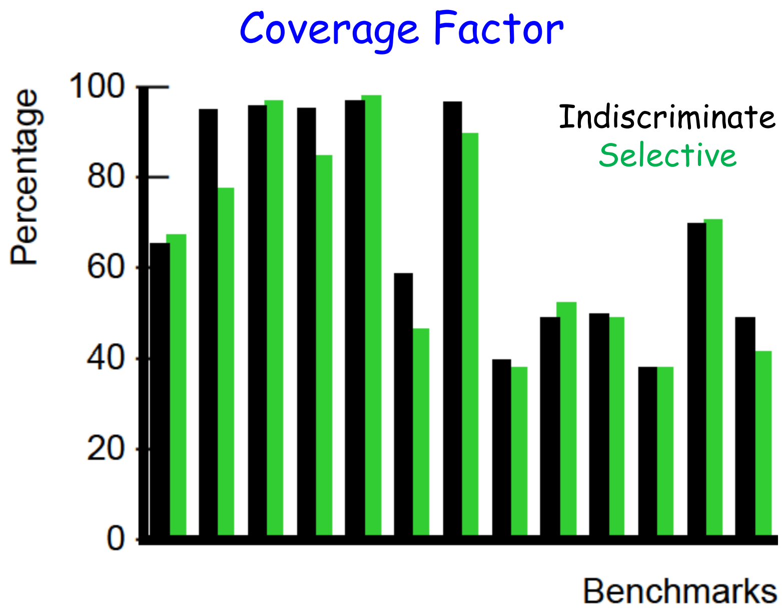
(**I** = Indiscriminate Prefetching, **S** = Selective Prefetching)

Selective vs. Indiscriminate prefetching:

- similar reduction in memory stalls
- significantly less overhead
- 6 of 13 have speedups over 20%

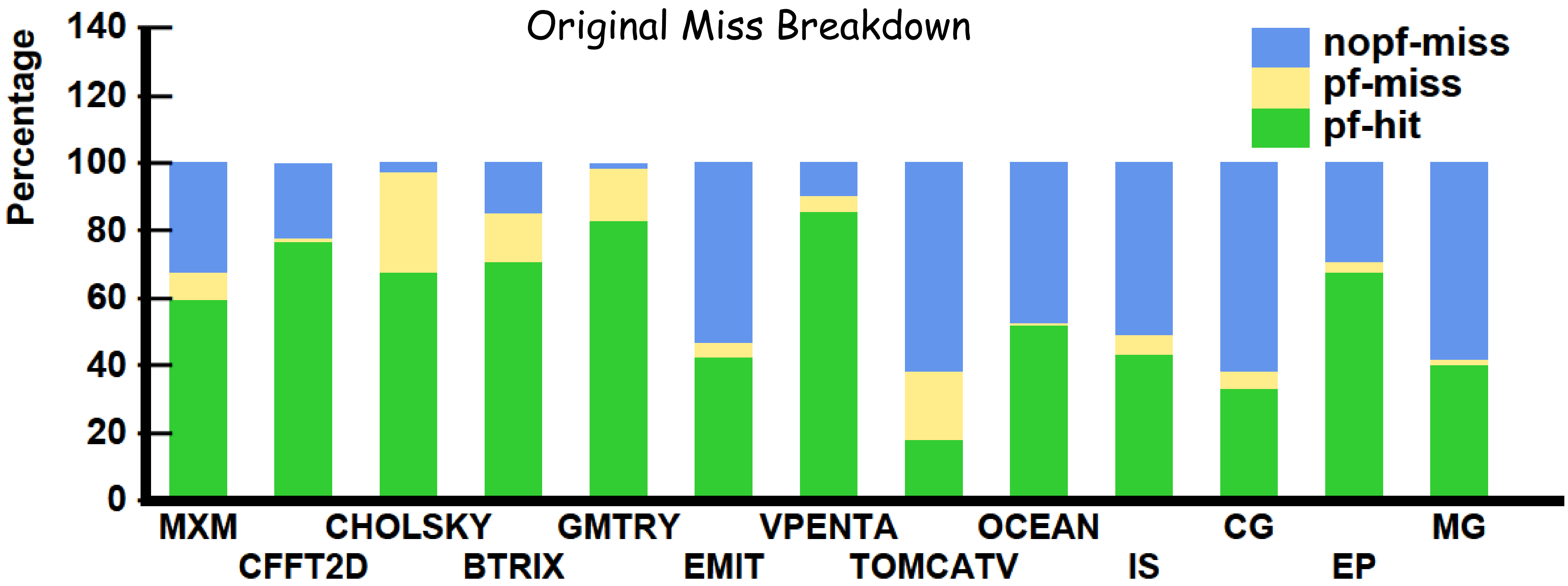
Effectiveness of Locality Analysis (Continued)





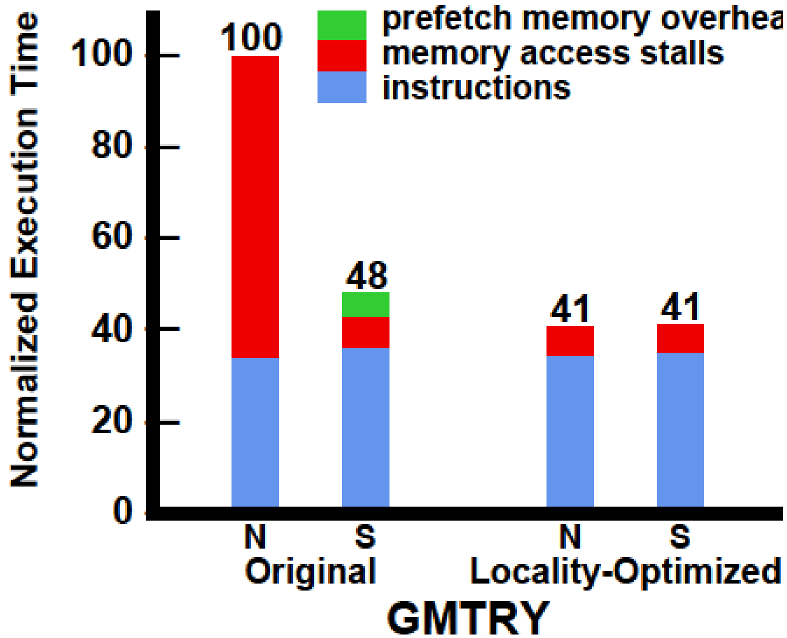
- fewer unnecessary prefetches
- comparable coverage factor
- reduction in prefetches ranges from 1.5 to 21 (average = 6)

Effectiveness of Software Pipelining

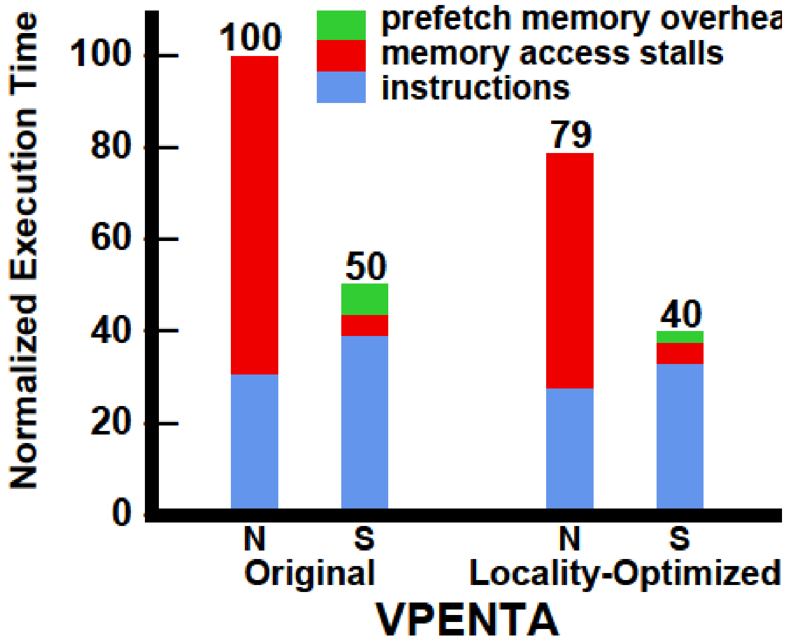


- Large pf-miss → ineffective scheduling
  - conflicts replace prefetched data (CHOLSKY, TOMCATV)
  - prefetched data still found in secondary cache

Interaction with Locality Optimizer



(Cache Blocking)



(Loop Interchange)

- locality optimizations reduce number of cache misses
- prefetching hides any remaining latency
- best performance through a combination of both

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

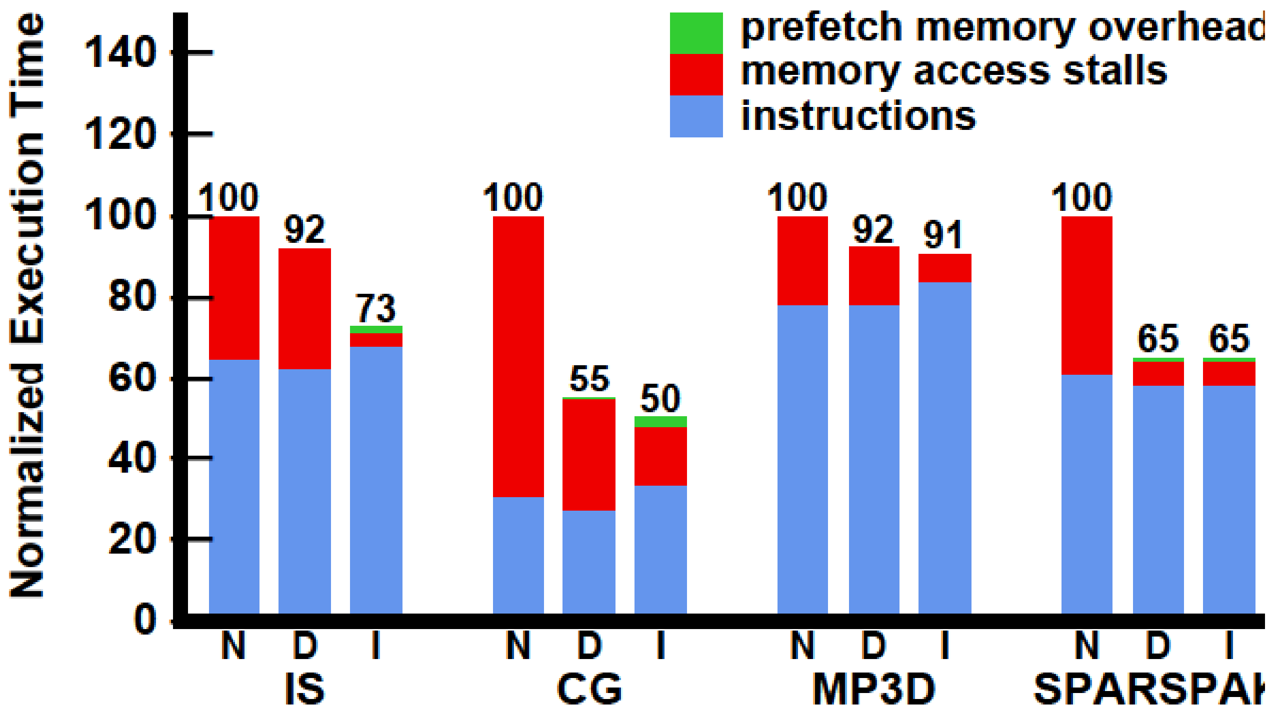
```
for (i = 0; i<100; i++)
   sum += A[index[i]];
```

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]];
```

Indirection Prefetching Results



(**N** = No Prefetching, **D** = Dense-Only Prefetching, **I** = Indirection Prefetching)

- larger overheads in computing indirection addresses
- significant overall improvements for IS and CG

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

Part II: 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

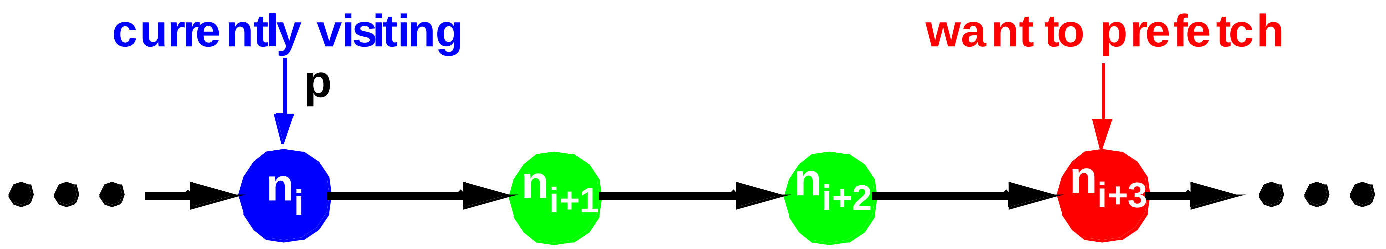
Goal:

- Automatic Compiler-Based Prefetching for Recursive Data Structures

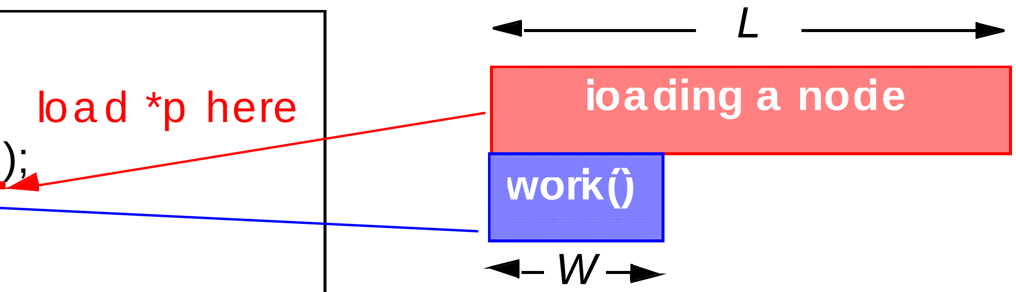
Overview

- 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;
}
```

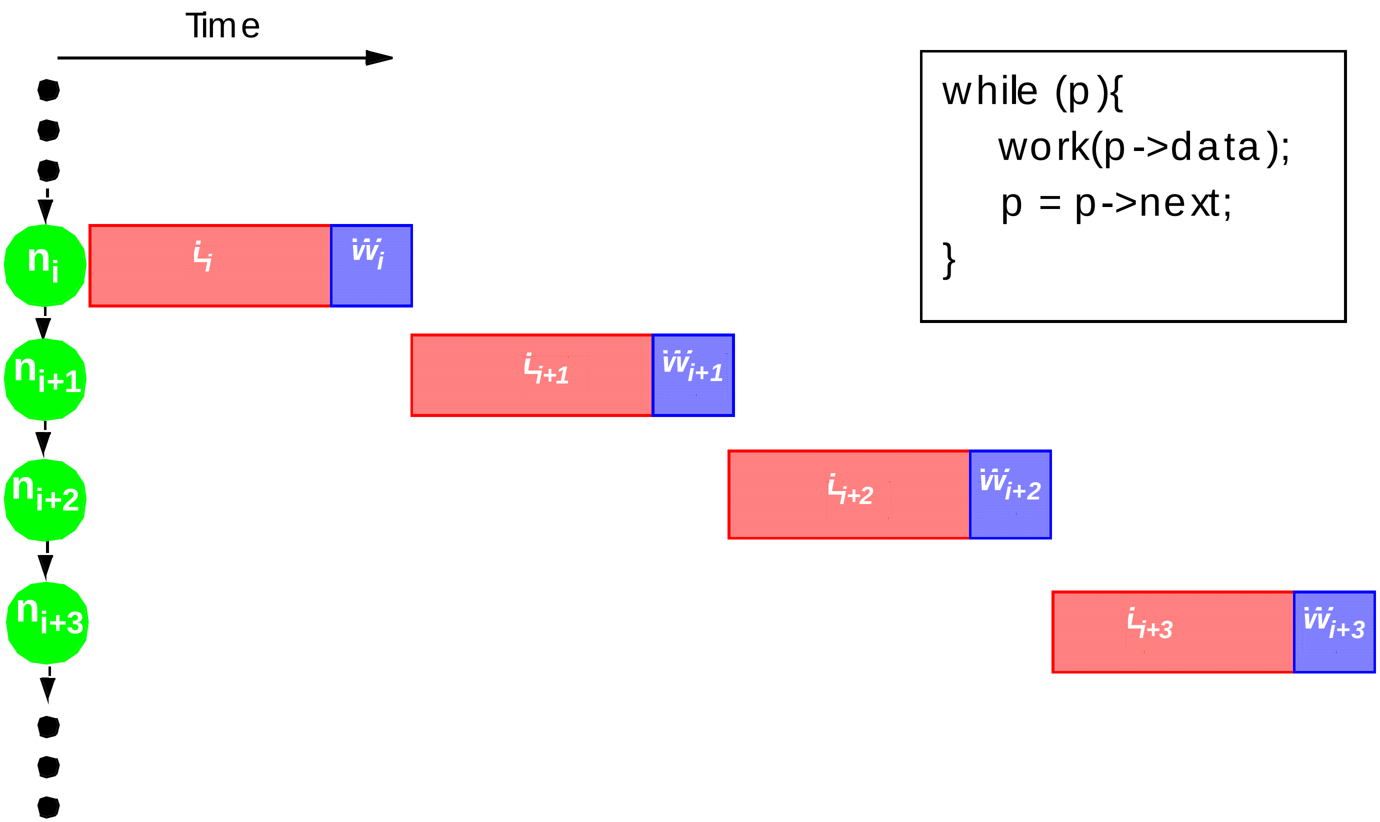


Our Goal: *fully hide latency*

- thus achieving fastest possible computation rate of 1/W

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

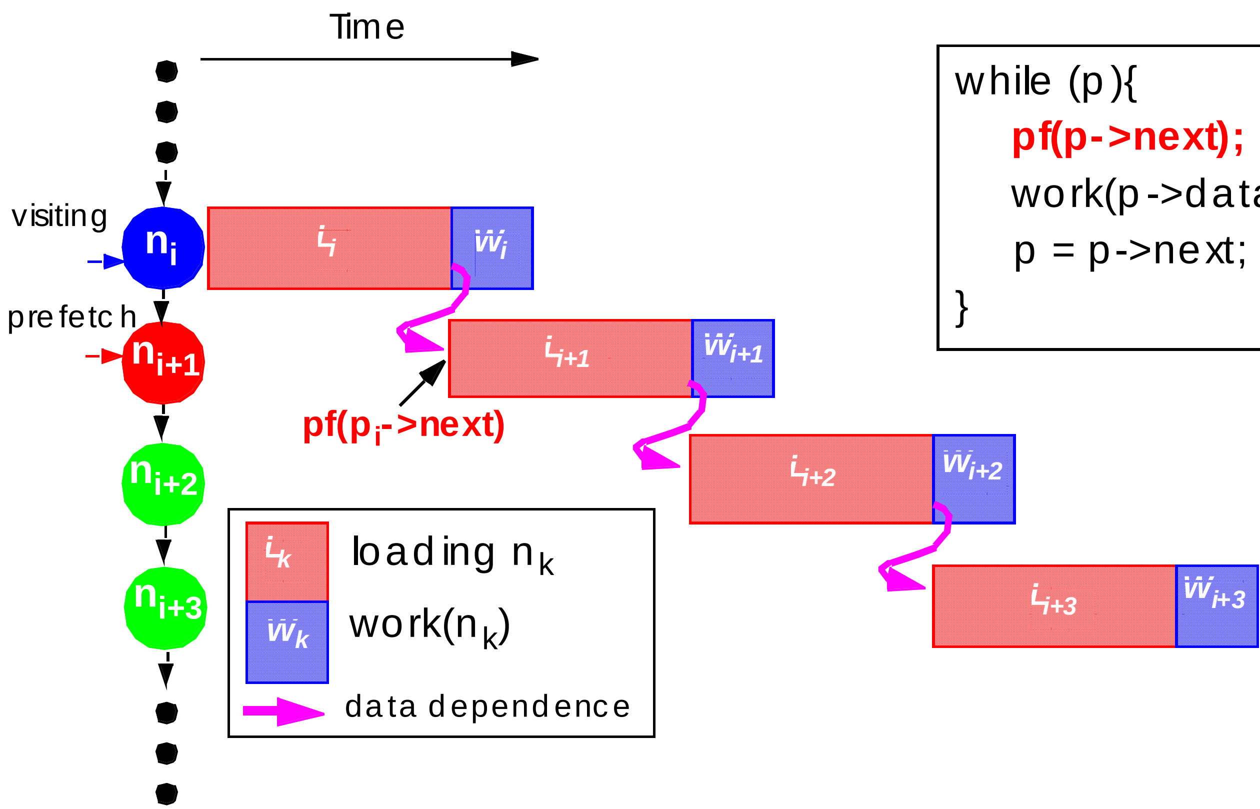
Performance without Prefetching




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

computation rate = 1 / (L+W)

Prefetching One Node Ahead



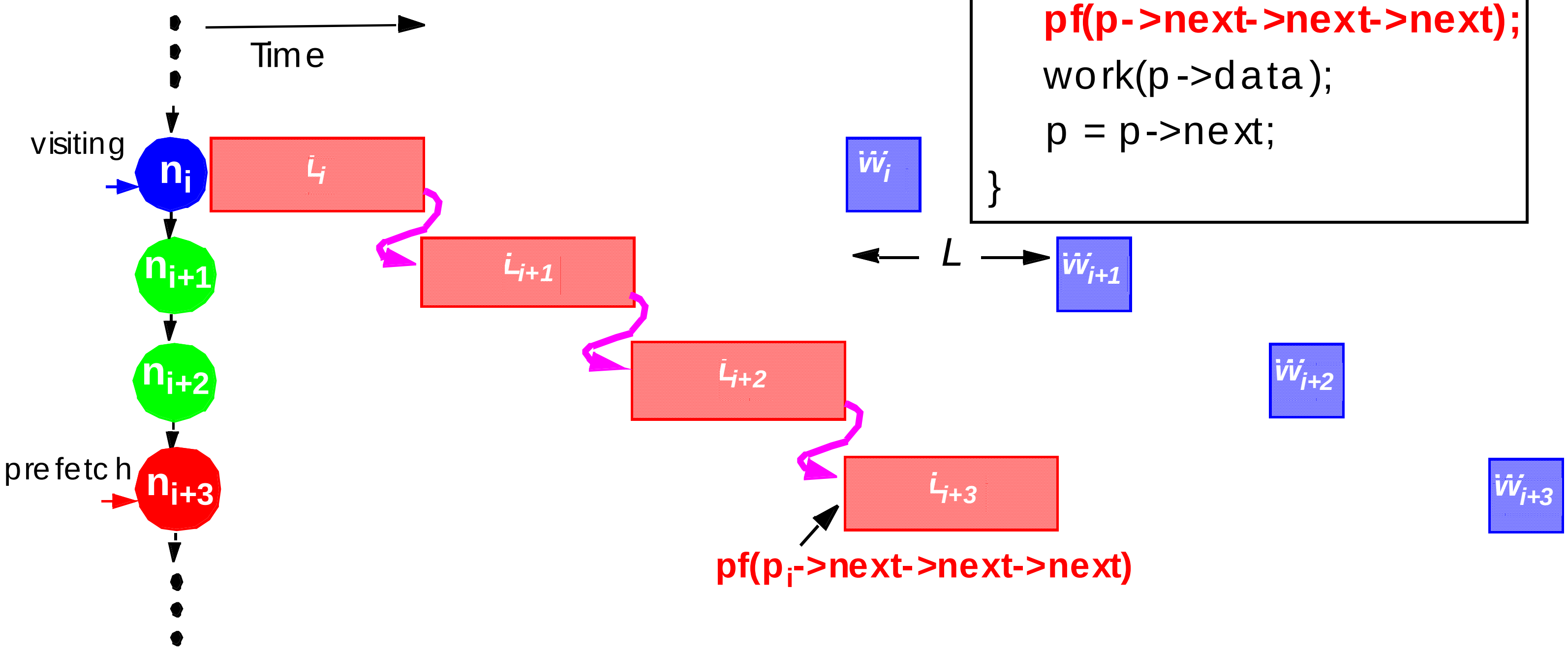

```
while (p){
    pf(p->next);
    work(p->data);
    p = p->next;
}
```

- 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;
}
```

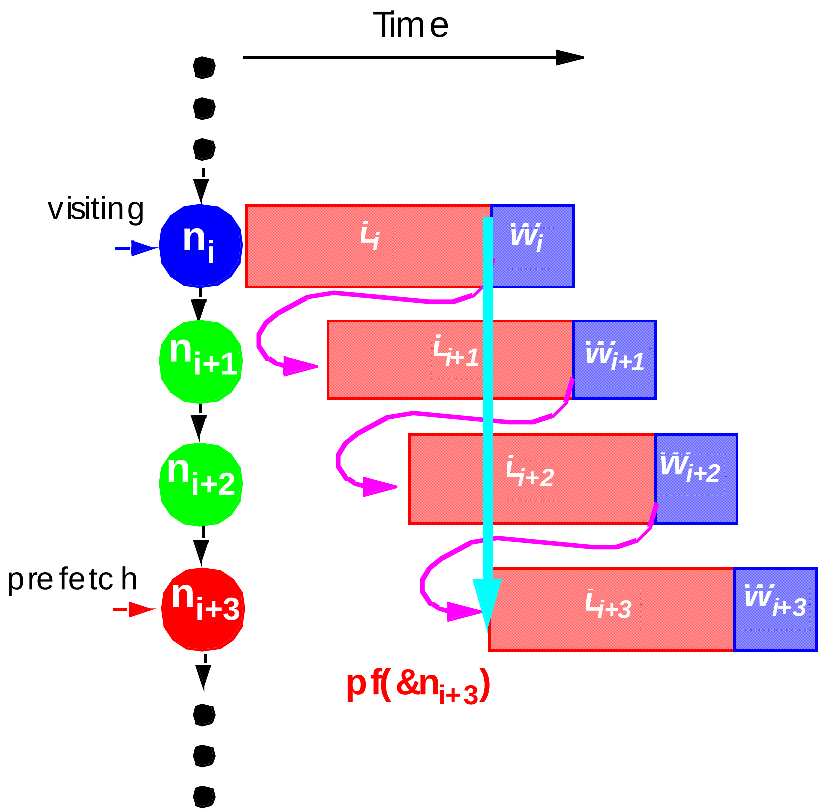


computation rate does not improve (still = 1/L)!

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

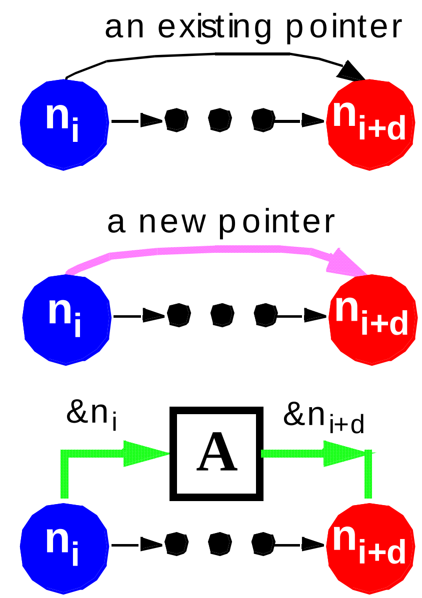
Overcoming the Pointer-Chasing Problem

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

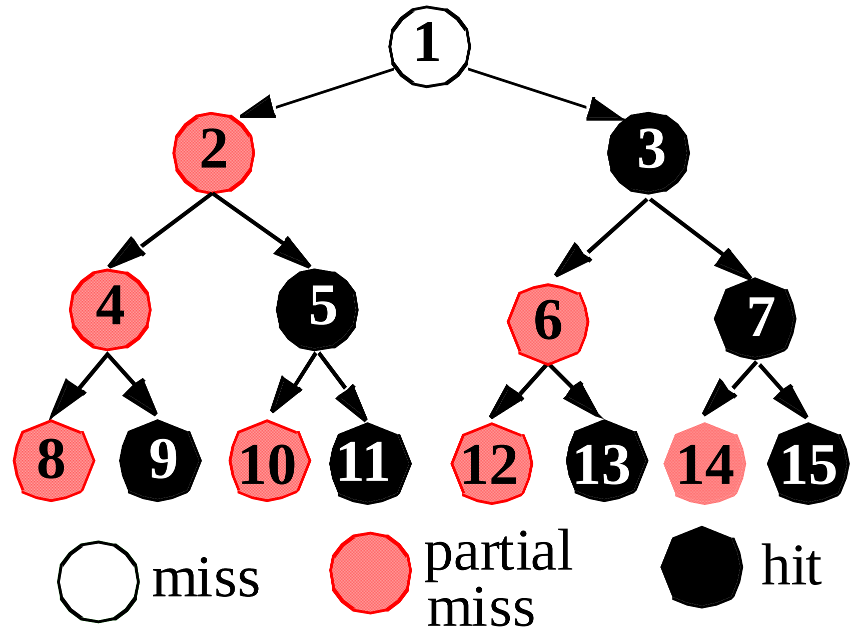


A=Address generating function

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

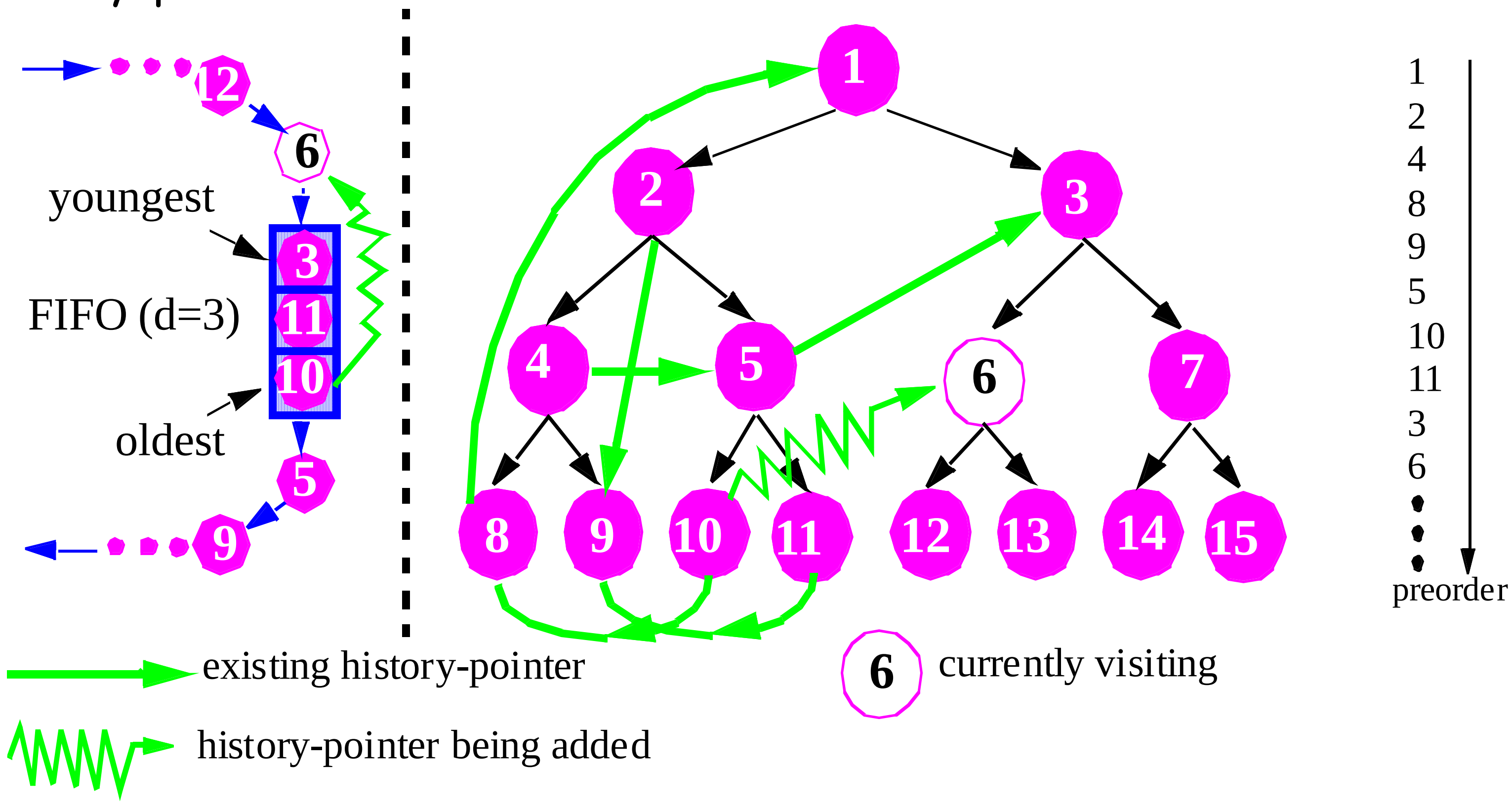
```
preorder(treeNode * t){
  if (t != NULL){
    pf(t->left);
    pf(t->right);
    process(t->data);
    preorder(t->left);
    preorder(t->right);
  }
}
```



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

History-Pointer Prefetching

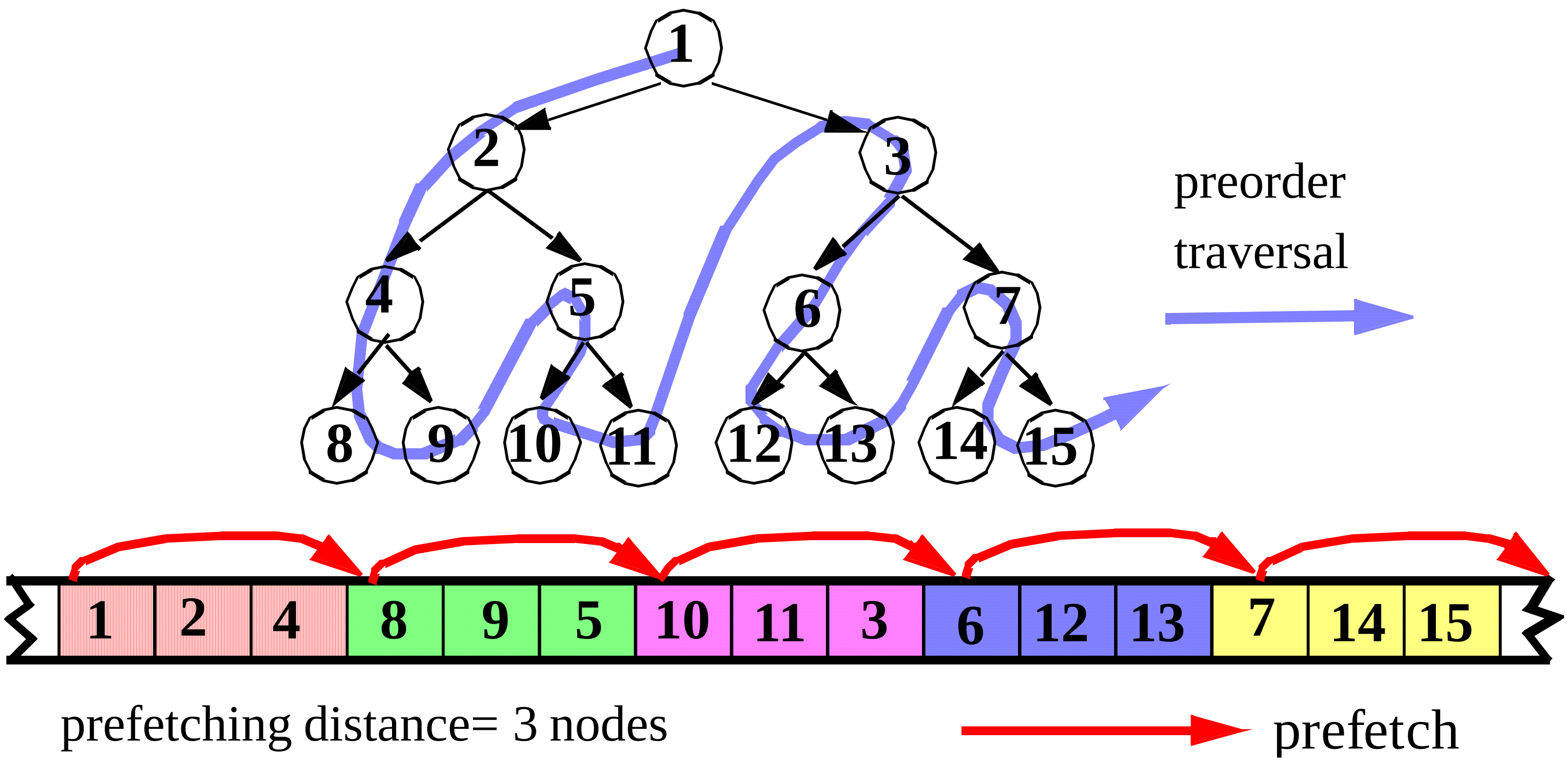
- Add new pointer(s) to each node
  - history-pointers are obtained from some recent traversal



- 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 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

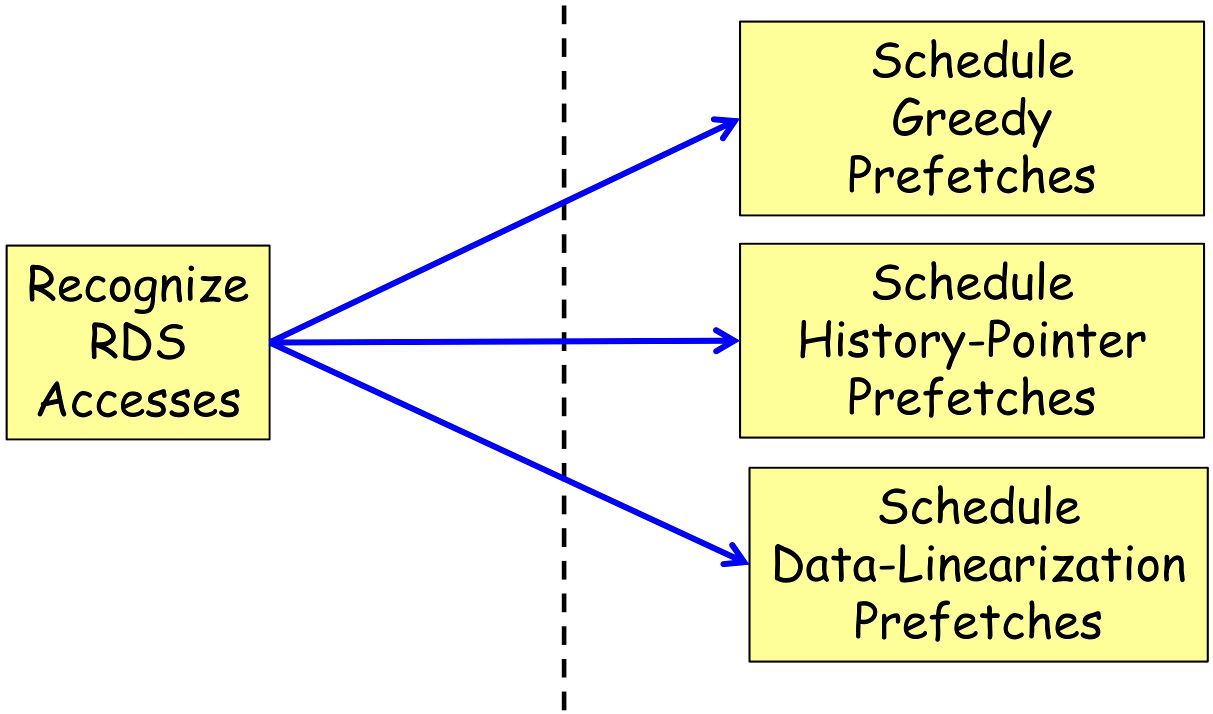
- Detailed, cycle-by-cycle simulations
- MIPS R10000-like dynamically-scheduled superscalar

Compiler

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

Implementation of Our Prefetching Algorithms

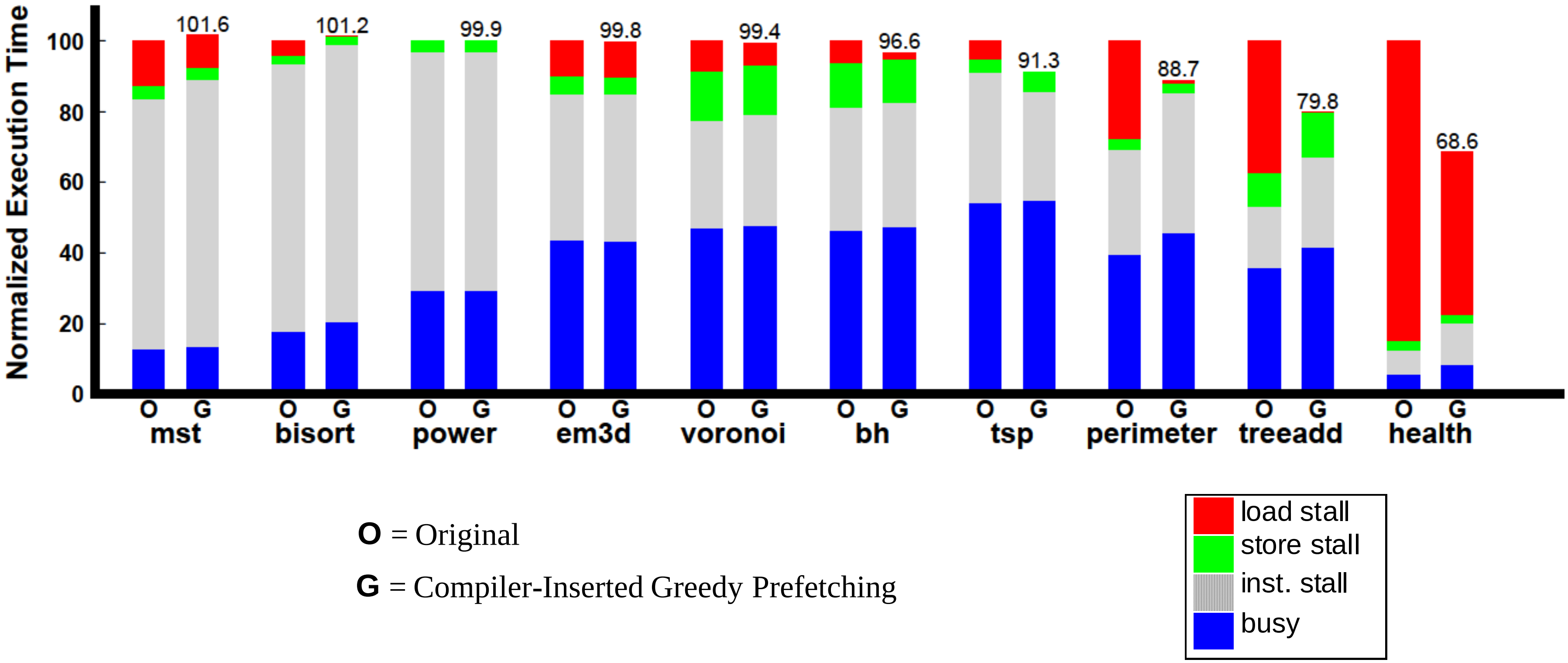
Automated in the SUIF compiler



- 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

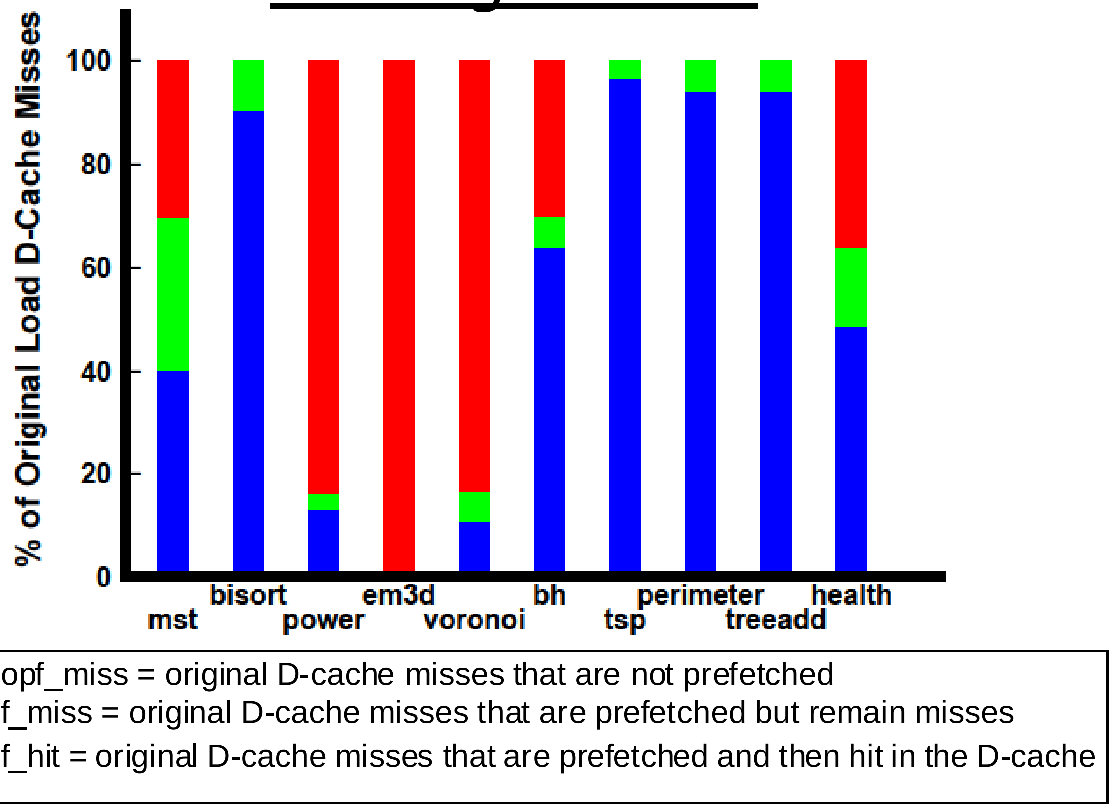


O = Original

G = Compiler-Inserted Greedy Prefetching

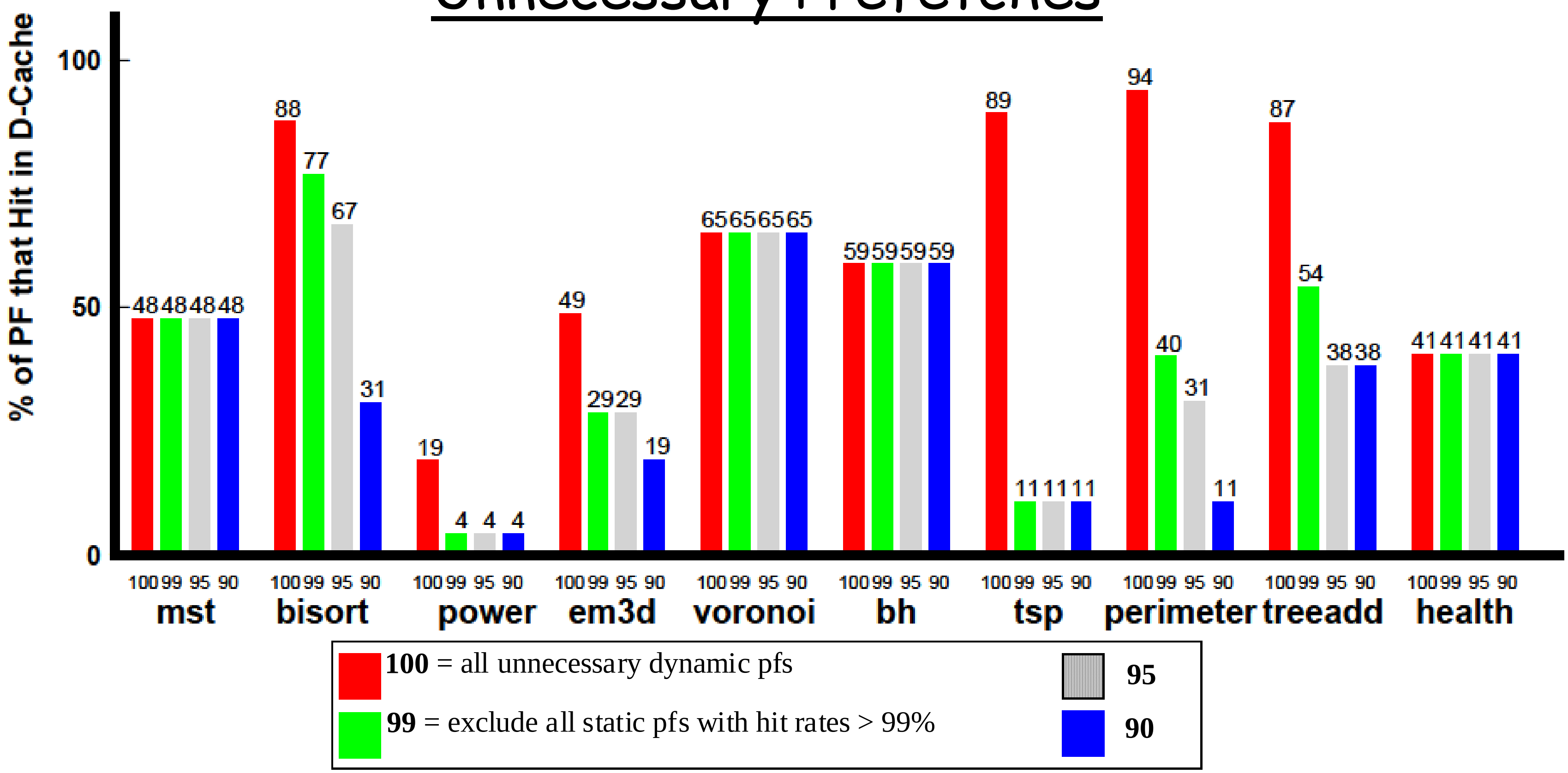
- Eliminates much of the stall time in programs with large load stall penalties
  - half achieve speedups of 4% to 45%

Coverage Factor



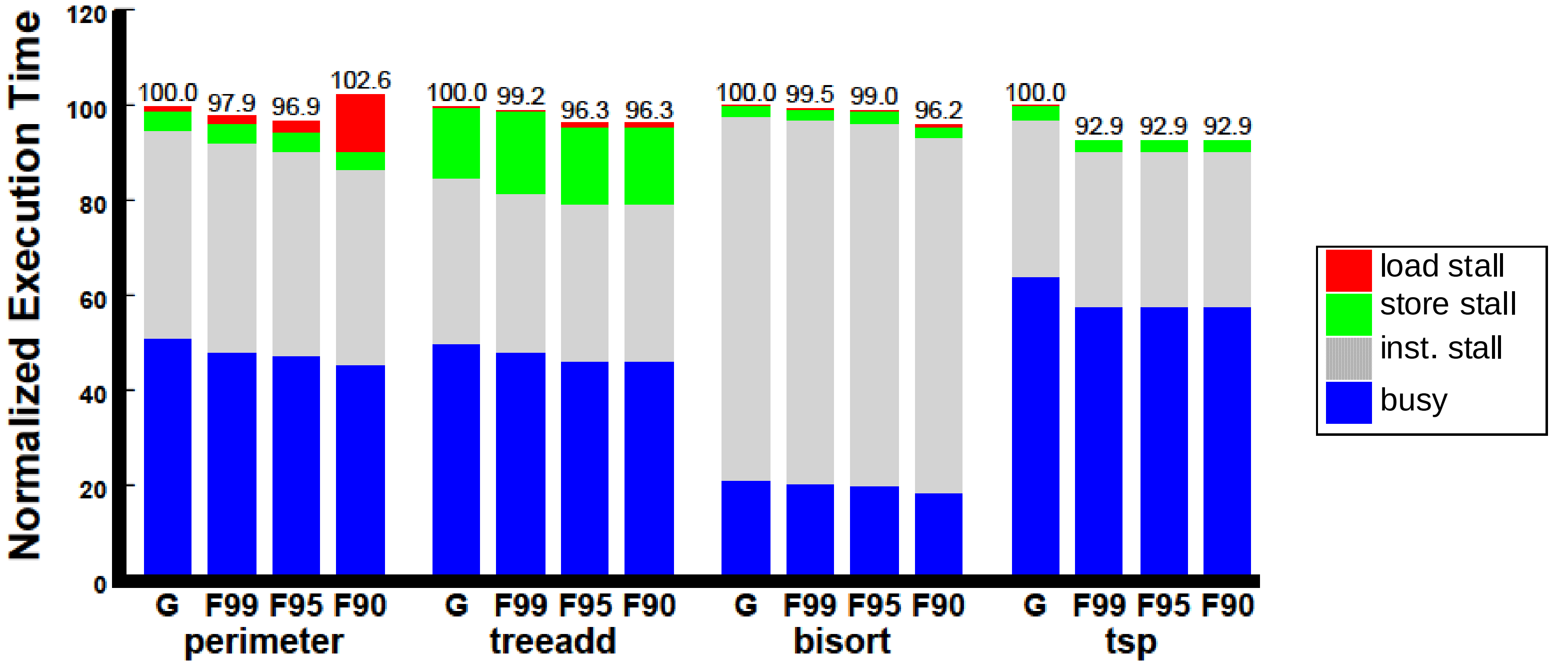
- coverage factor = pf_hit + pf_miss
- 7 out of 10 have coverage factors > 60%
  - em3d, power, voronoi have many array or scalar load misses
- small pf_miss fractions → effective prefetch scheduling

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

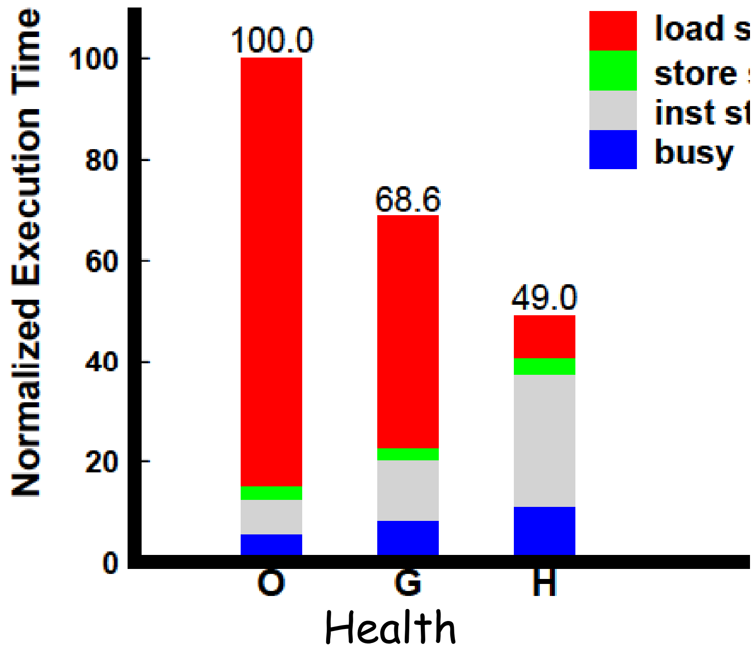


G = greedy prefetching
Fxx = greedy prefetching where static pfs with hit rate > xx% are eliminated

- 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

Carnegie Mellon

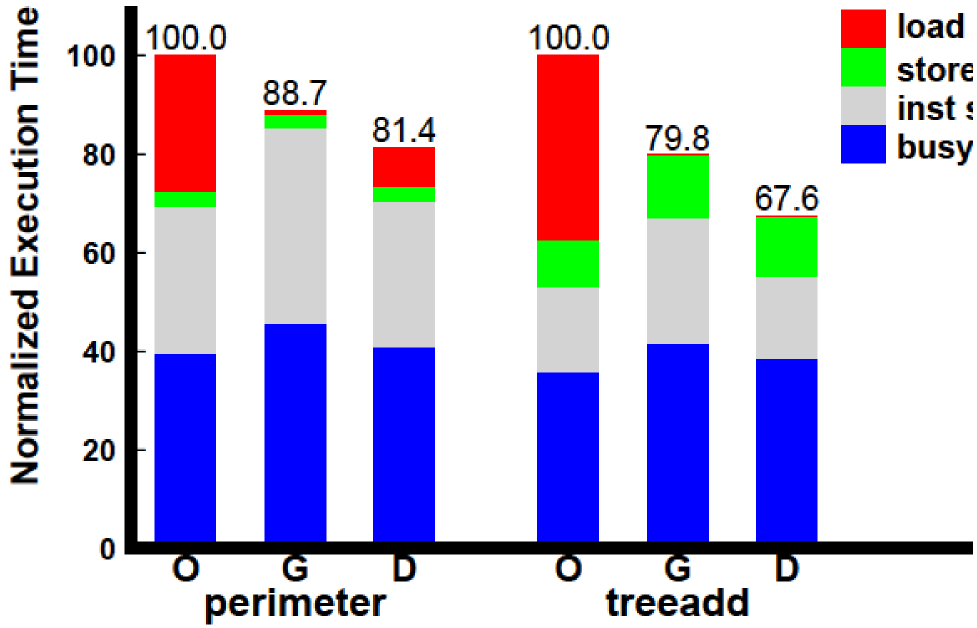
Performance of History-Pointer Prefetching



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

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