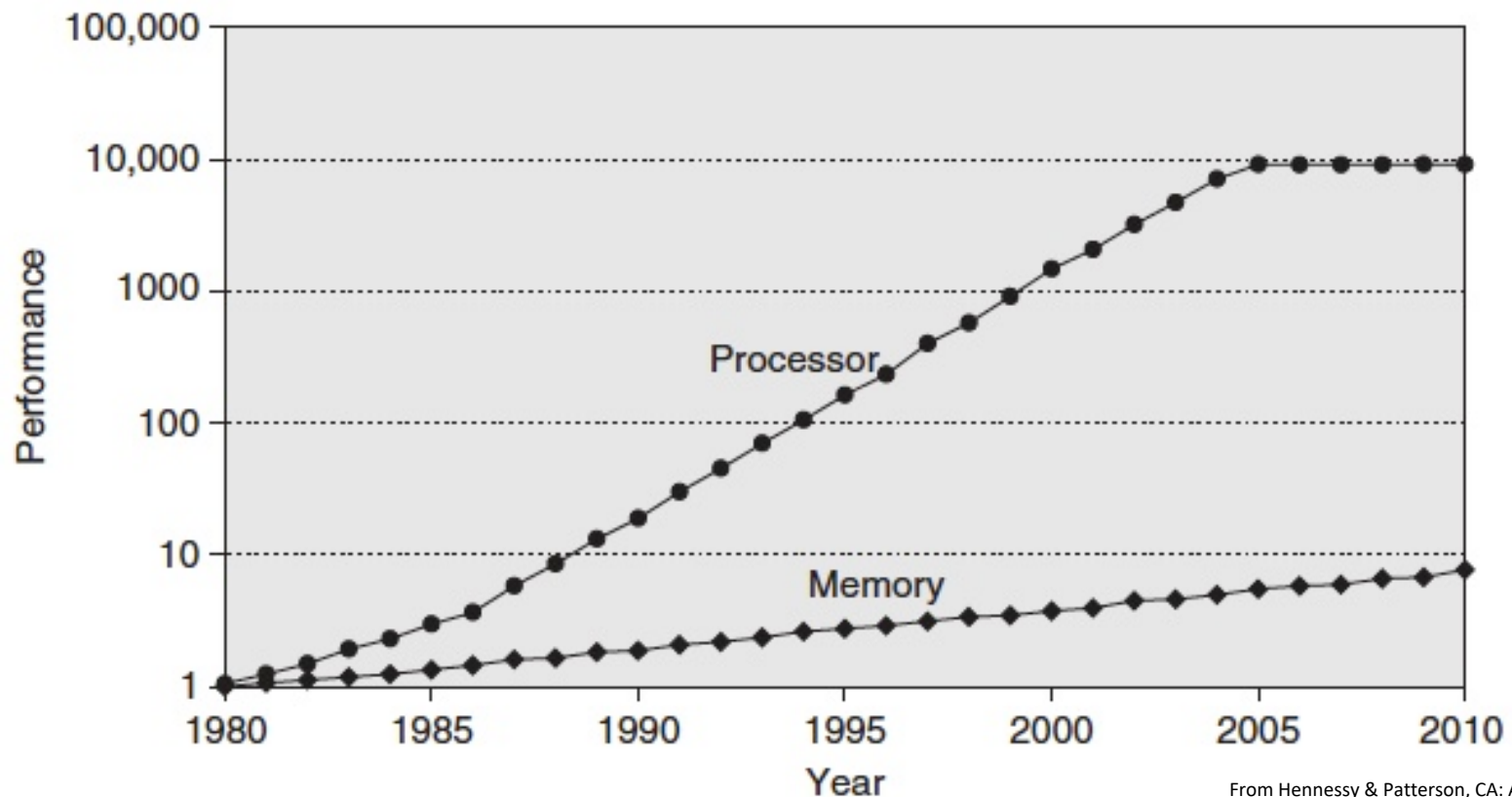


Tolerating Latency Through Prefetching

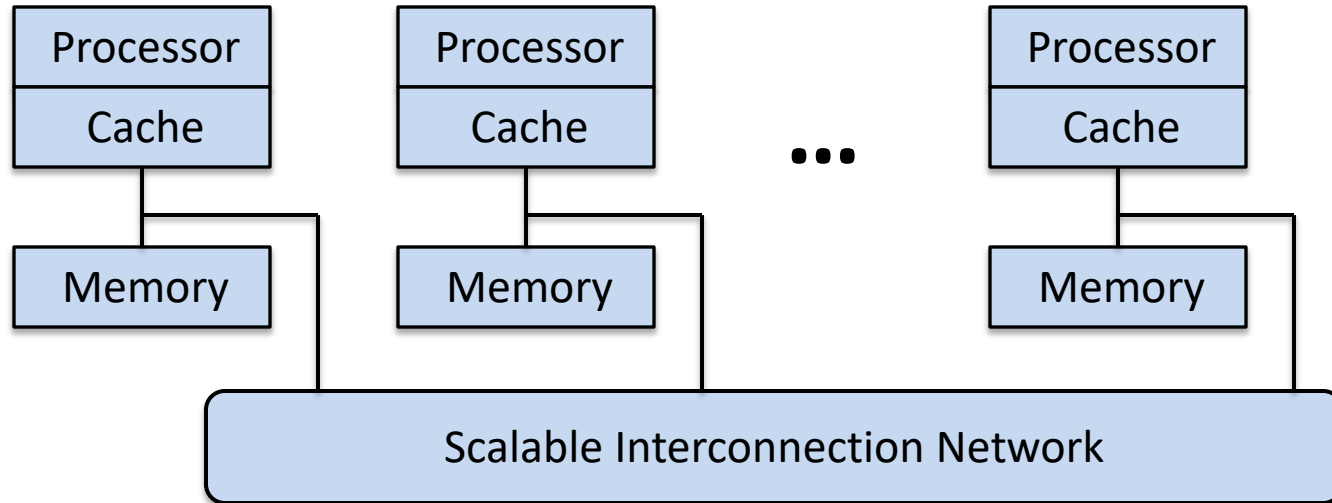
Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2019

The Memory Latency Problem



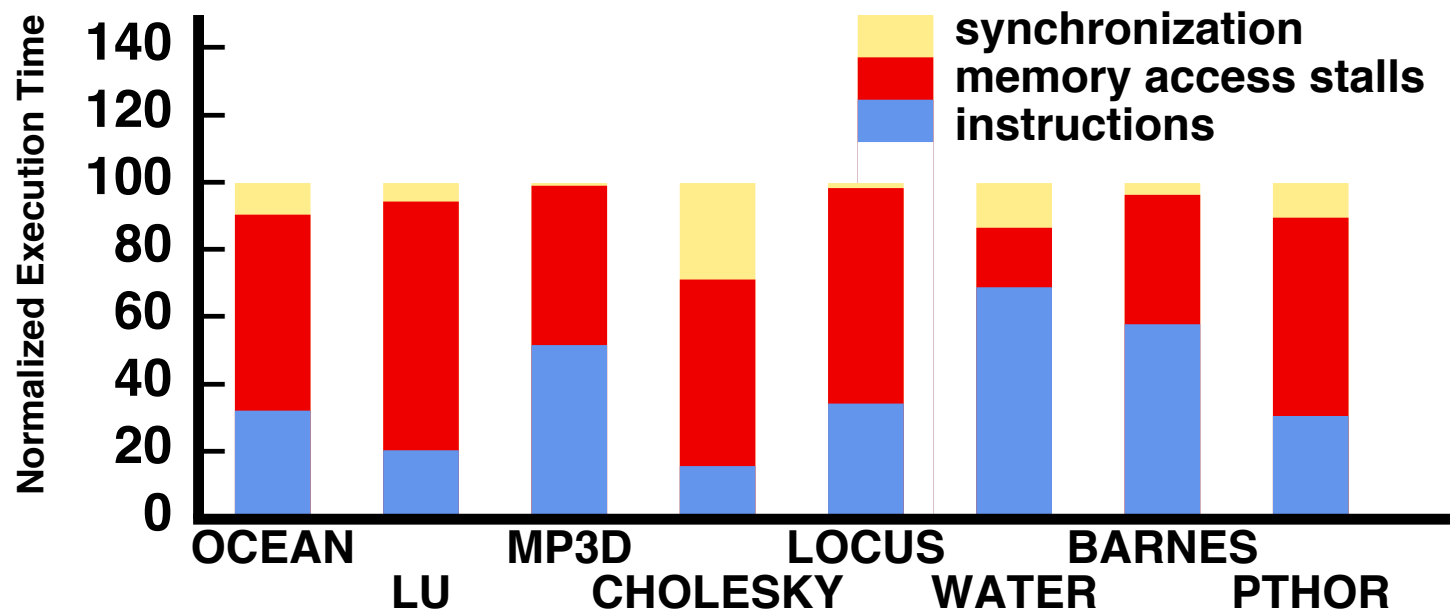
- \uparrow processor speed \gg \uparrow memory speed
- caches help, but are not a panacea

Even Worse: Remote Latencies in a NUMA Multiprocessors



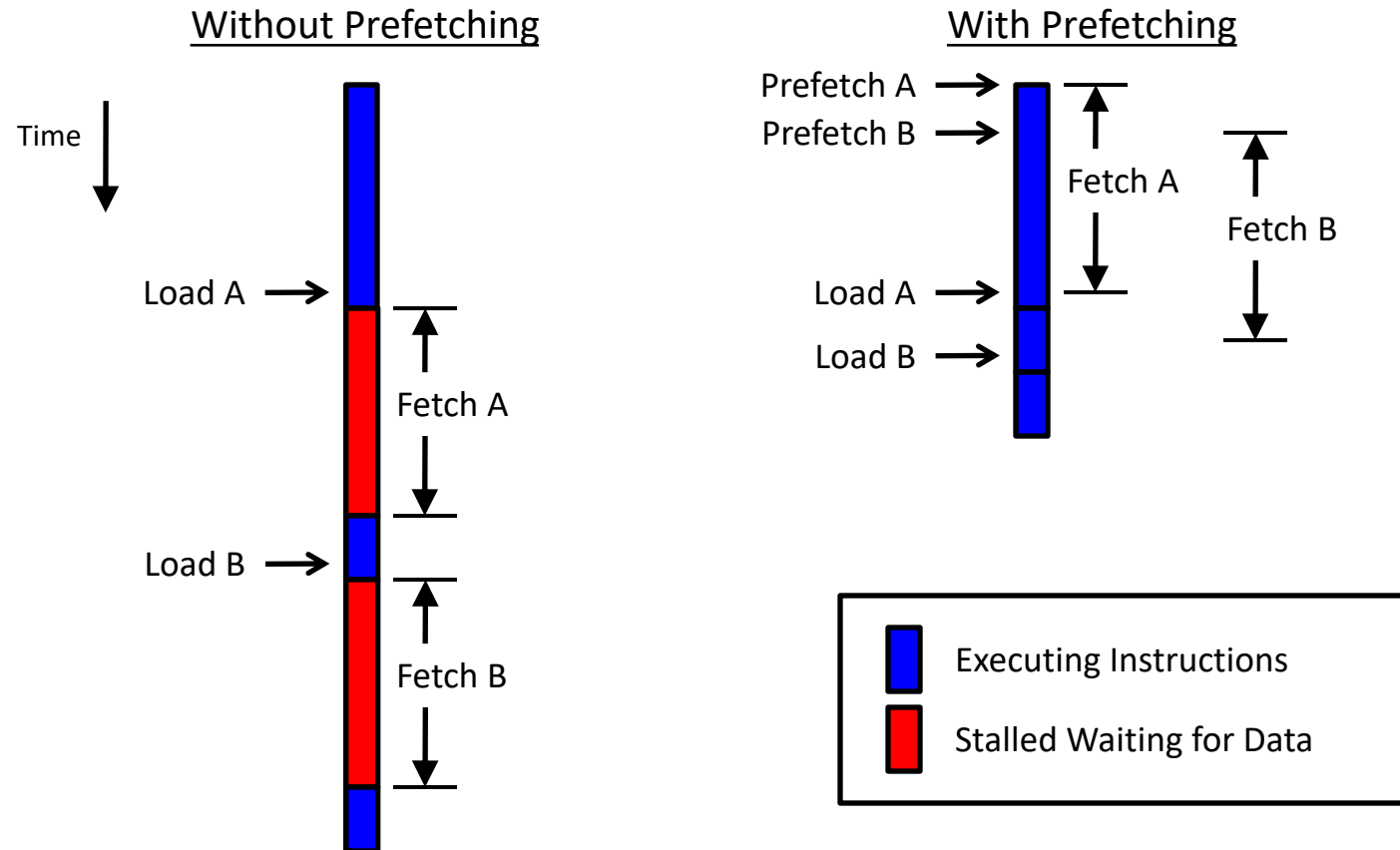
- Long cache miss latencies due to:
 - remote memory accesses
 - cache coherence

Impact of Memory Latency in these Parallel Machines



- 16-processor shared-address space machine (similar to DASH multiprocessor)
 - directory-based cache coherence
 - latencies = 1 : 15 : 30 : 100 : 130 processor cycles
- 6 of 8 spend > 50% of time stalled for memory!

Tolerating Latency Through Prefetching



- overlap memory accesses with computation and other accesses

Benefits of Prefetching

- Prefetch early enough
 - completely hides memory latency
- Issue prefetches in blocks
 - pipeline the misses
 - only the first reference stalls
- Prefetch with ownership
 - reduces write latency, coherence messages

Types of Prefetching

- Large cache blocks
 - limitations: spatial locality, false sharing
- Hardware-controlled prefetching
 - modern processors detect (and prefetch) simple strided access patterns
 - limitations: simple patterns, page boundaries, potential cache pollution
- Software-controlled prefetching
 - explicit instructions in modern instruction sets
 - advantages: more sophisticated access patterns
 - limitations: instruction overhead?

Compiler-Based Prefetching

- How well can we do if we fully-automate prefetch insertion?
- What access patterns can we handle successfully?
 - arrays, pointers?
- Improving performance requires:
 - maximizing benefit, while
 - minimizing 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

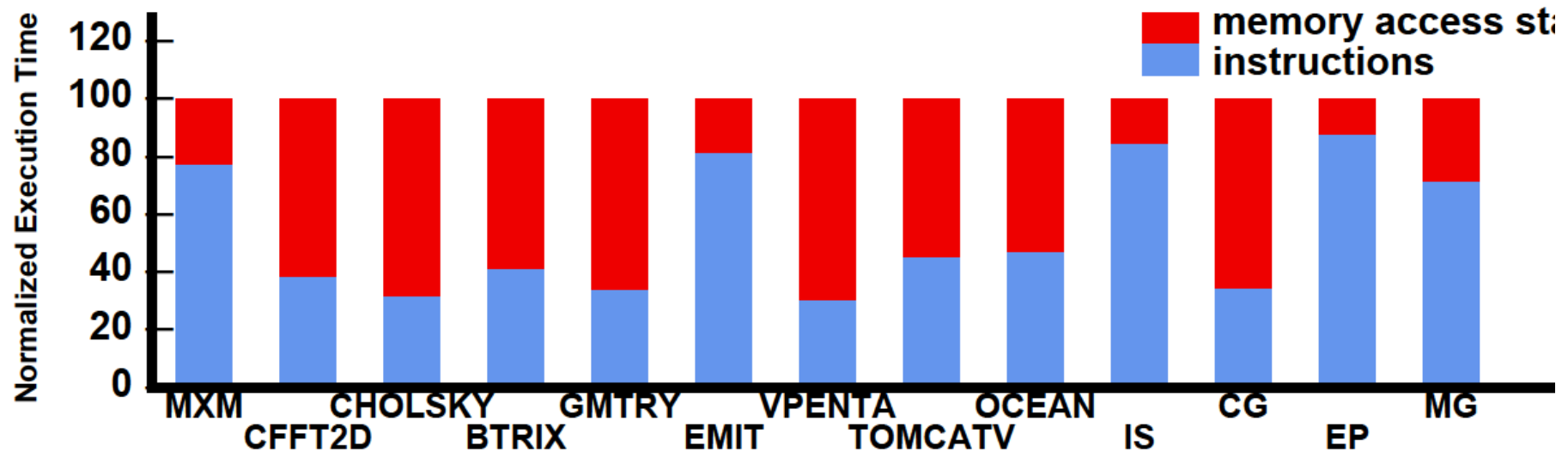
Analysis: what to prefetch

- maximize coverage factor
- minimize unnecessary prefetches

Scheduling: when/how to schedule prefetches

- maximize effectiveness
- minimize overhead per prefetch

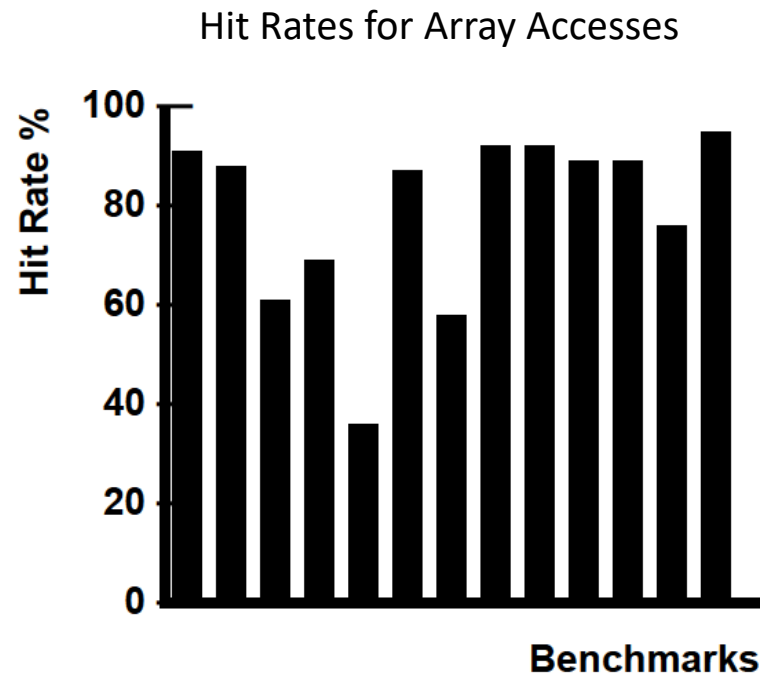
Let's Start with Prefetching for Sequential Applications



- 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

Reducing Prefetching Overhead

- instructions to issue prefetches
- extra demands on memory system



- 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”

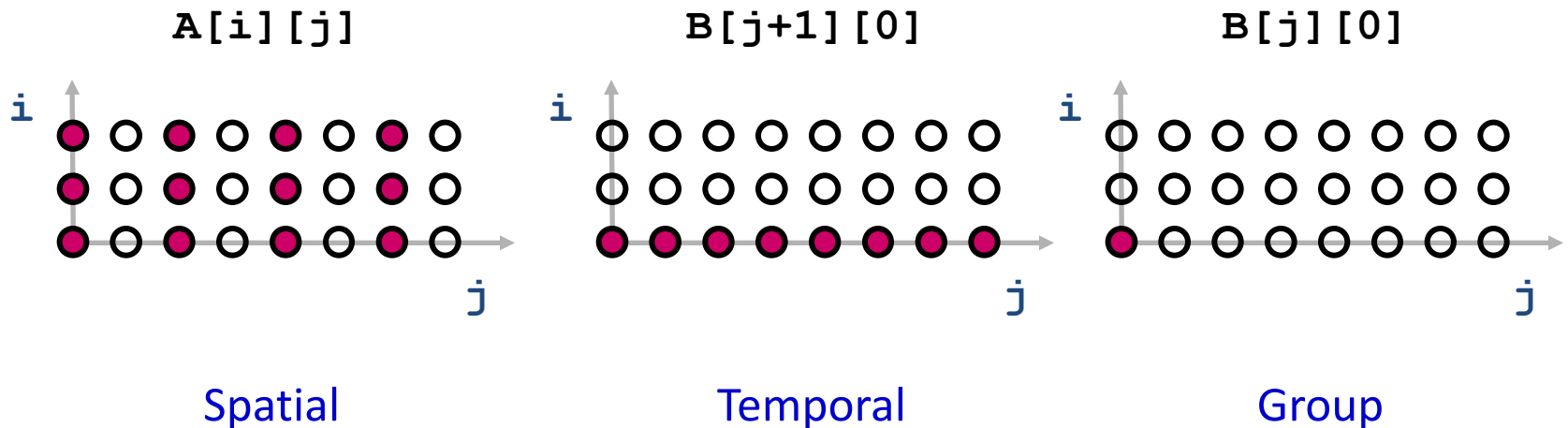
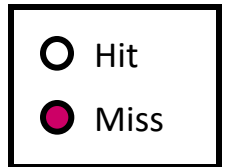
- set of inner loops where the data accessed by an iteration is expected to fit within the cache

3. Find data locality:

- $\text{reuse} \cap \text{localized iteration space} \Rightarrow \text{locality}$

Data Locality Example

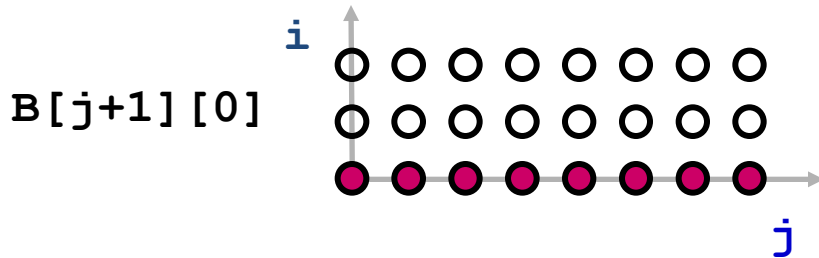
```
for i = 0 to 2
  for j = 0 to 100
    A[i][j] = B[j][0] + B[j+1][0];
```



Localized Iteration Space

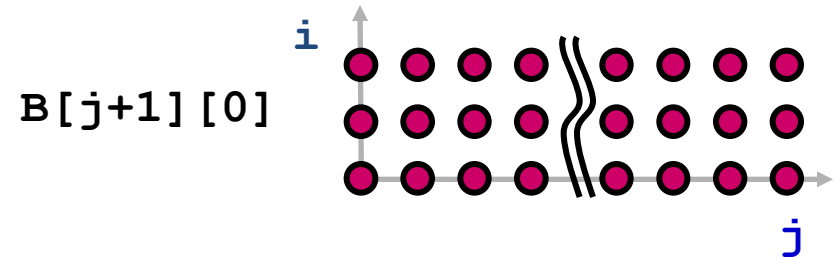
- Given finite cache, **when does reuse result in locality?**

```
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

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



Localized: j loop only

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

Prefetch Predicate

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

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	Loop Transformation
None	None
Temporal	Peel loop i
Spatial	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)

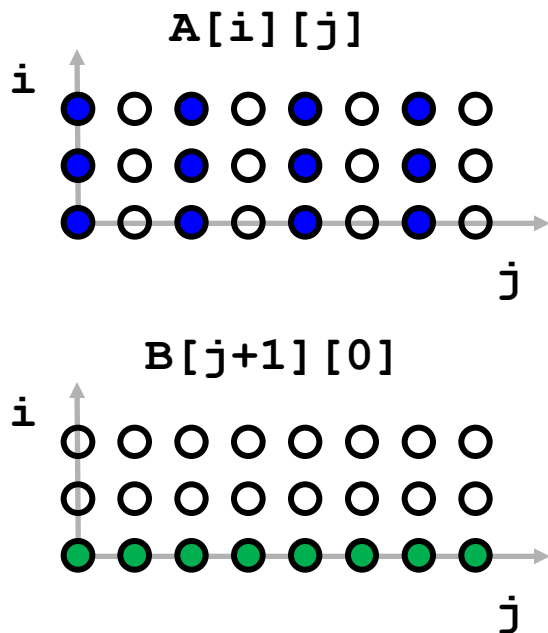
```
for (i = 0; i < 5; i++)          /* Prolog */  
    prefetch(&a[i]);  
  
for (i = 0; i < 95; i++) { /* Steady State */  
    prefetch(&a[i+5]);  
    a[i] = 0;  
}  
  
for (i = 95; i < 100; i++) /* Epilog */  
    a[i] = 0;
```

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

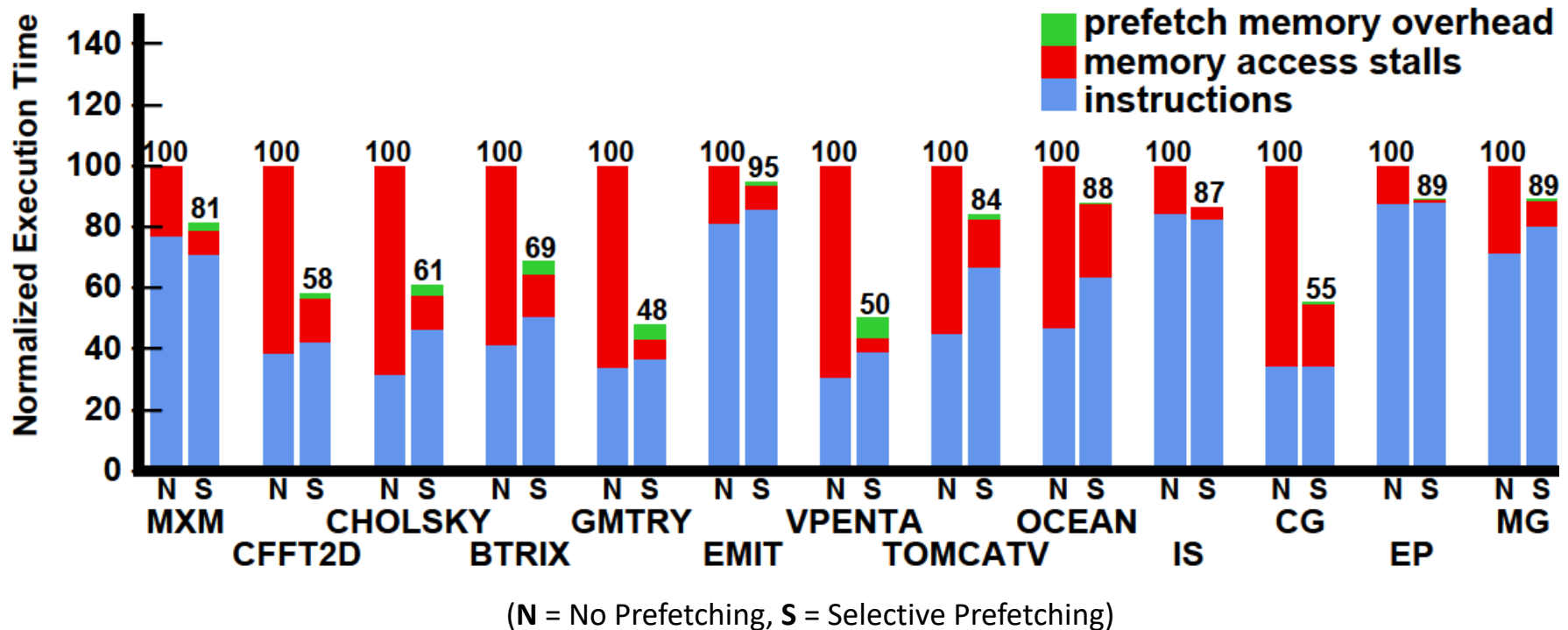


Code with Prefetching

```
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];
}

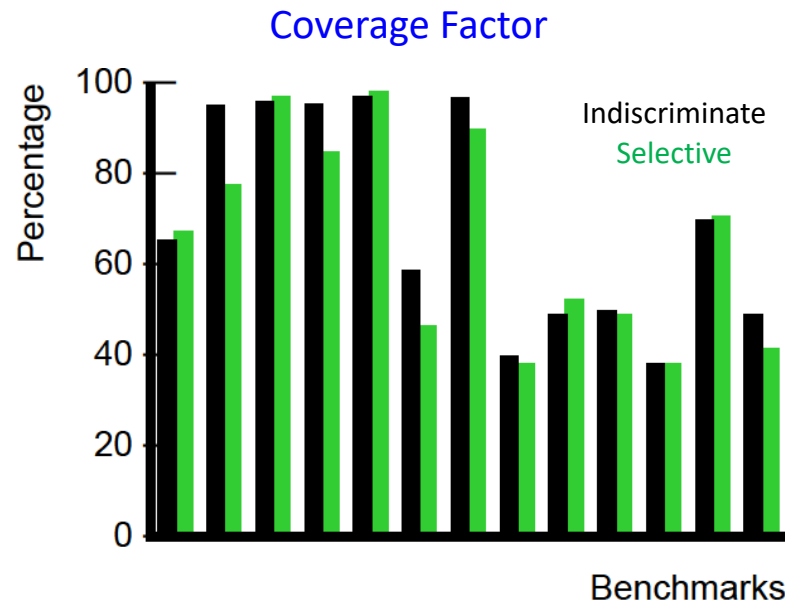
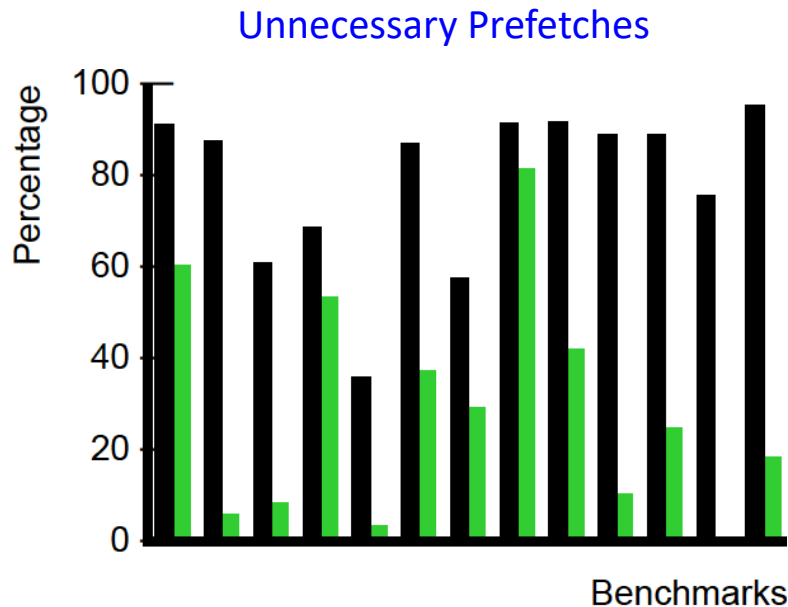
for (i = 1; i < 3; i++) {
  prefetch(&A[i][0]);
  for (j = 0; j < 6; j += 2)
    prefetch(&A[i][j+1]);
  for (j = 0; j < 94; j += 2) {
    prefetch(&A[i][j+7]);
    A[i][j] = B[j][0] + B[j+1][0];
    A[i][j+1] = B[j+1][0] + B[j+2][0];
  }
  for (j = 94; j < 100; j += 2) {
    A[i][j] = B[j][0] + B[j+1][0];
    A[i][j+1] = B[j+1][0] + B[j+2][0];
  }
}
```


Performance of Prefetching Algorithm



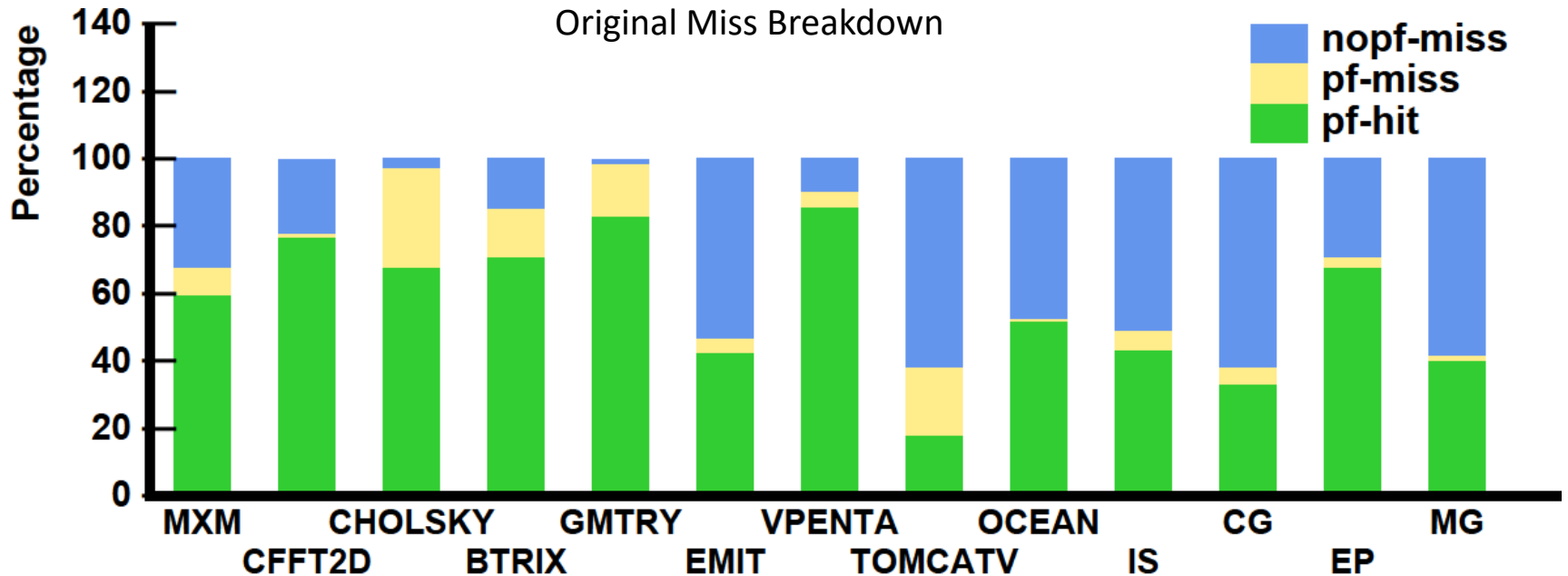
- memory stalls reduced by 50% to 90%
- instruction and memory overheads typically low
- 6 of 13 have speedups over 45%

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

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

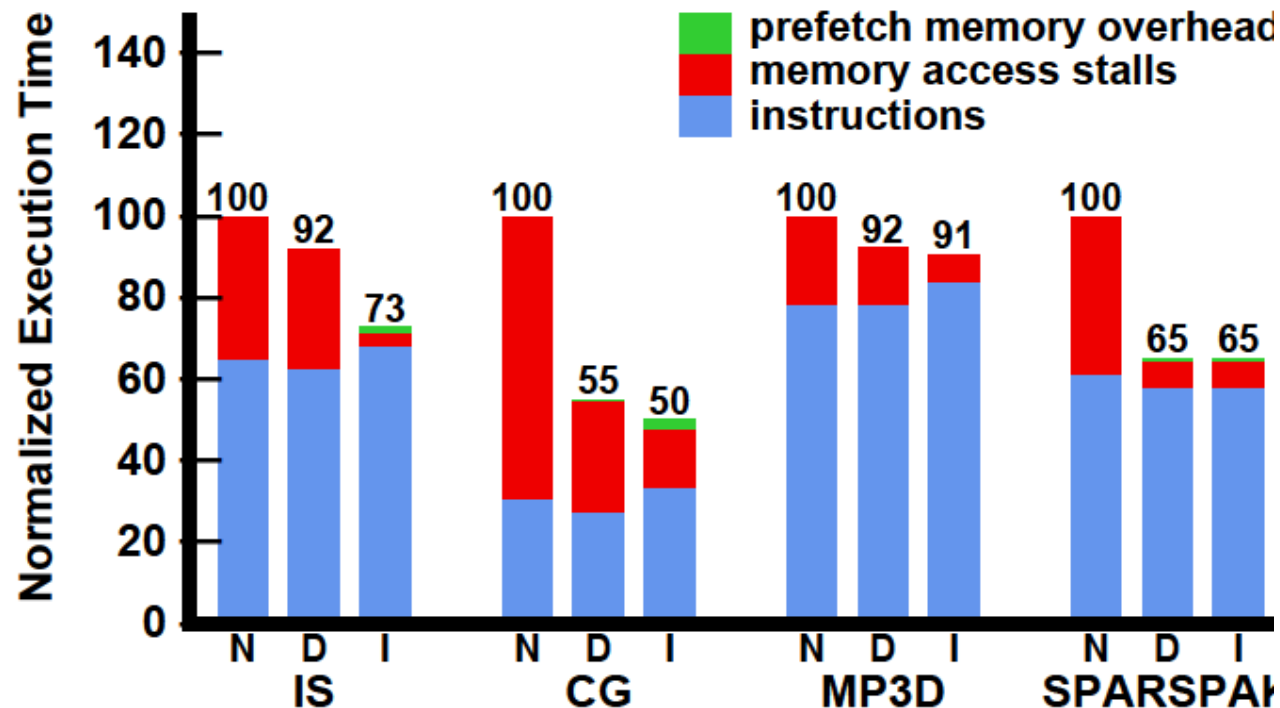
Original Loop

```
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]];
```

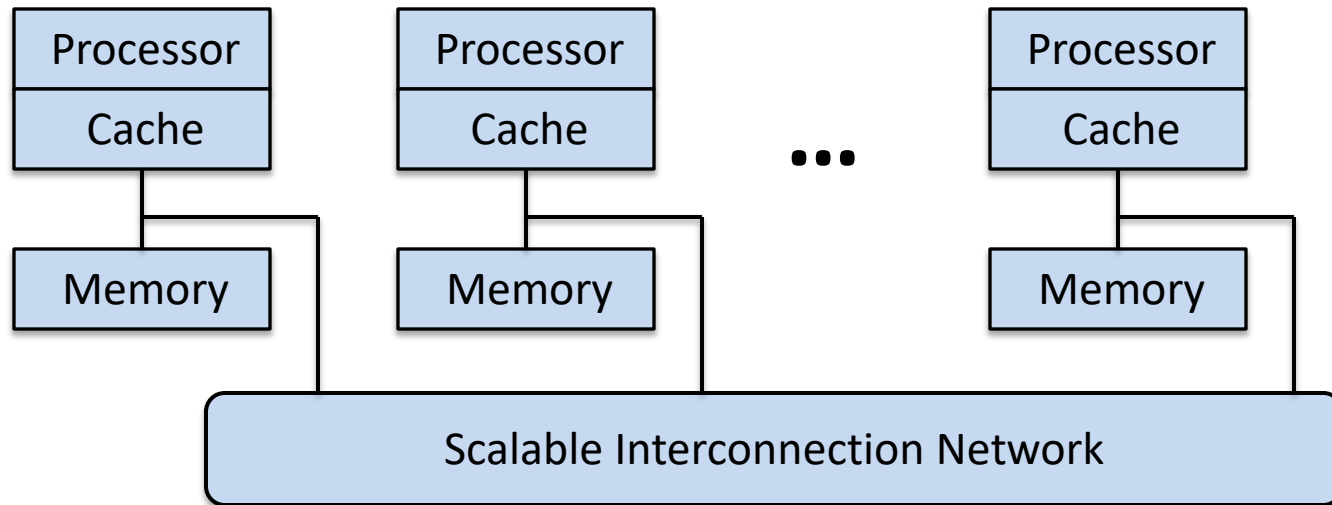
Indirection Prefetching Results



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

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

Prefetching for Parallel Shared-Address-Space Machines



- Main memory is *physically distributed* (aka **NUMA**)
 - but logically a single, *shared address space*
- Hardware cache coherence

Prefetching for Multiprocessors

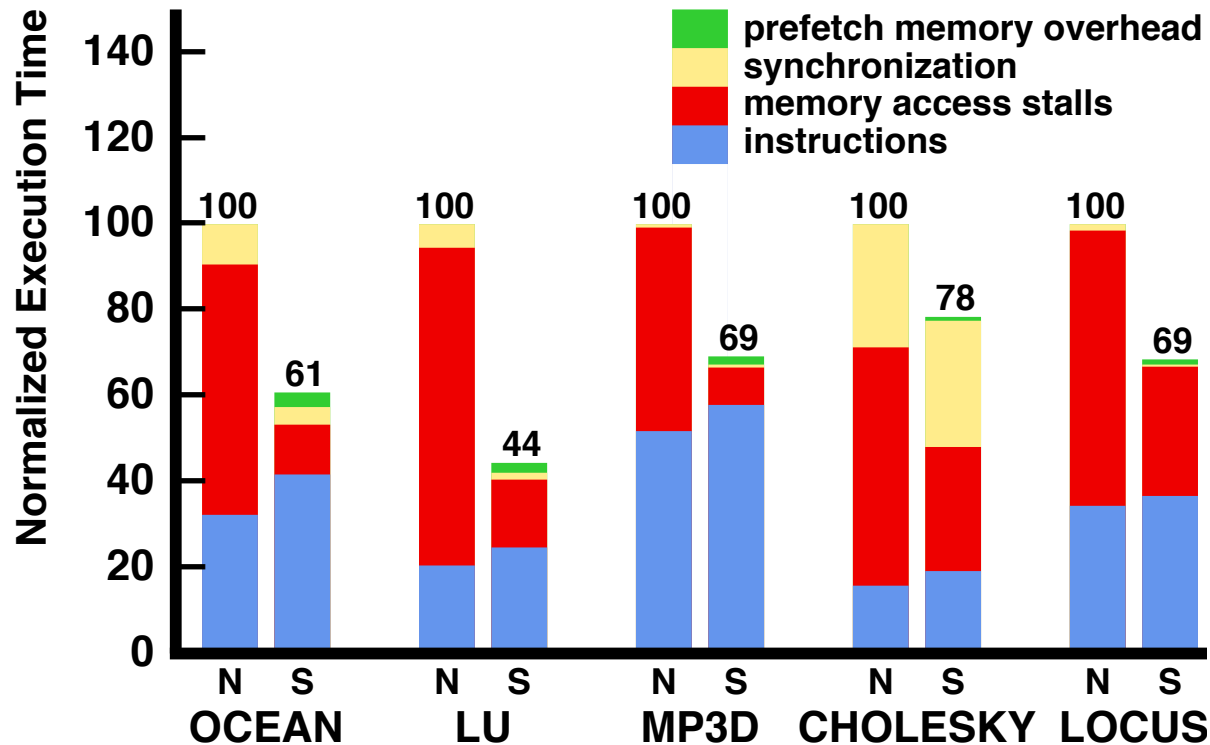
- *Non-binding vs. binding prefetches:*
 - use **non-binding** since data remains coherent until accessed later

```
prefetch (&x) ;  
...  
LOCK (L) ;  
x = x + 1 ;  
UNLOCK (L) ;
```

→ **no restrictions on when prefetches can be issued**

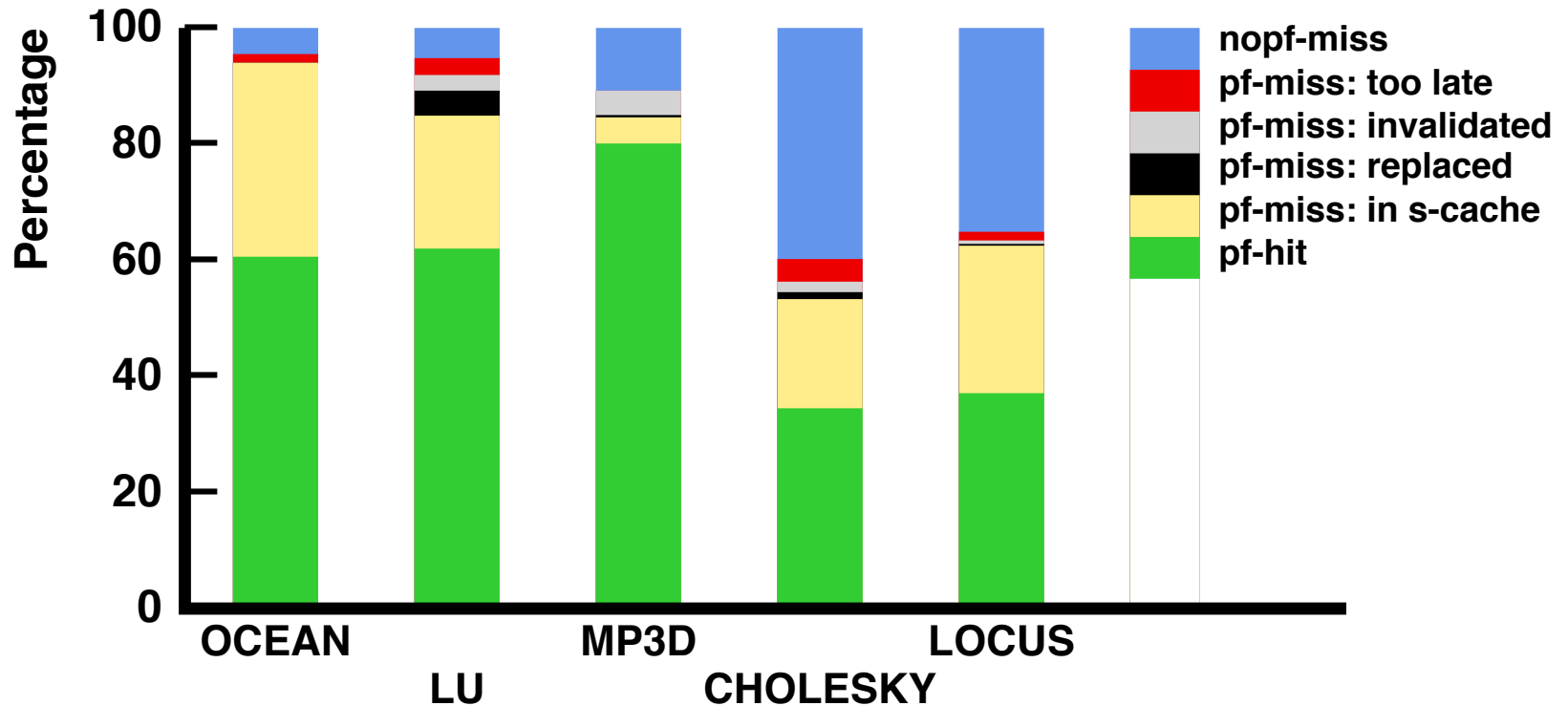
- *Dealing with coherence misses:*
 - localized iteration space takes explicit synchronization into account
- *Further optimizations:*
 - prefetching in **exclusive-mode** in **read-modify-write** situations

Multiprocessor Results



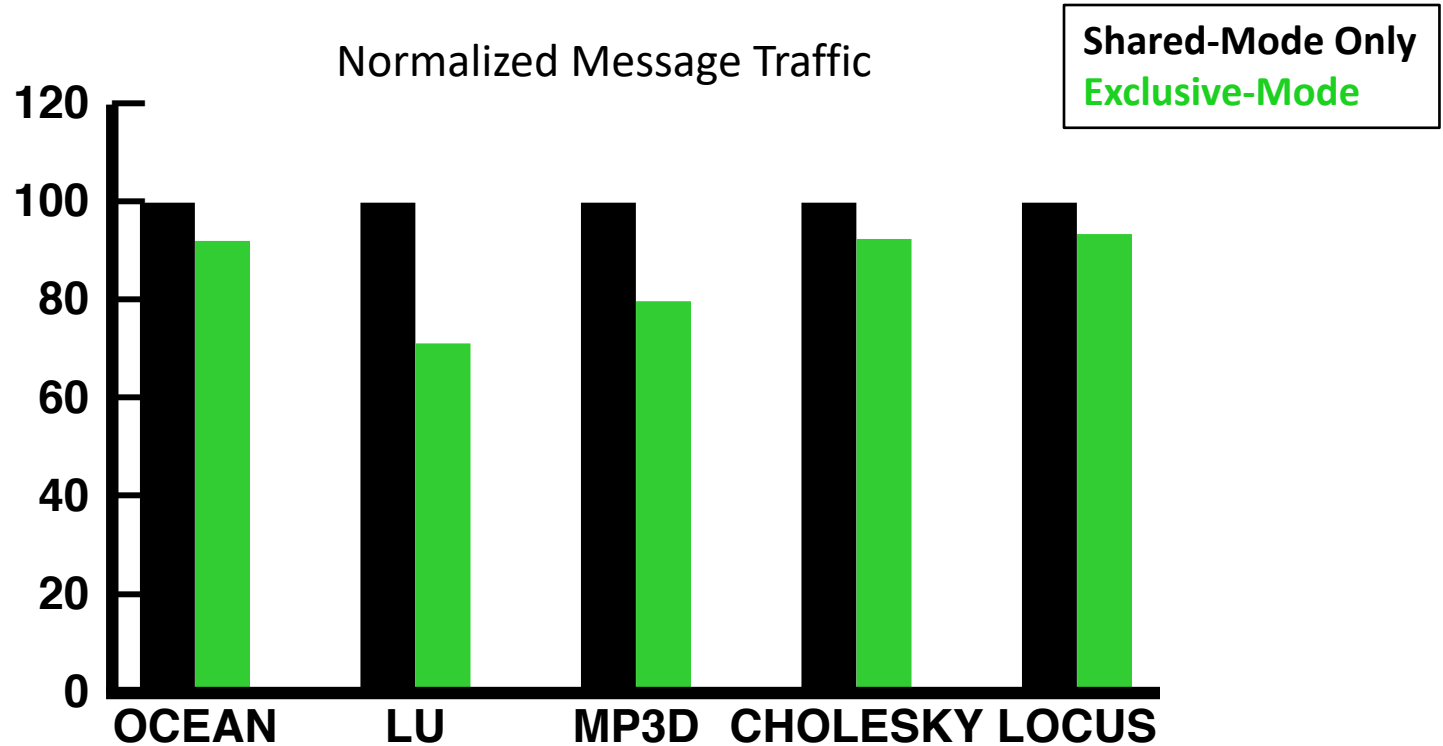
- Memory stalls reduced by 50% to 90%
 - Synchronization stalls reduced in some cases
- 4 of 5 have speedups over 45%

Effectiveness of Software Pipelining



- Large pf-miss → ineffective scheduling
 - prefetched data still found in secondary cache

Exclusive-Mode Prefetching



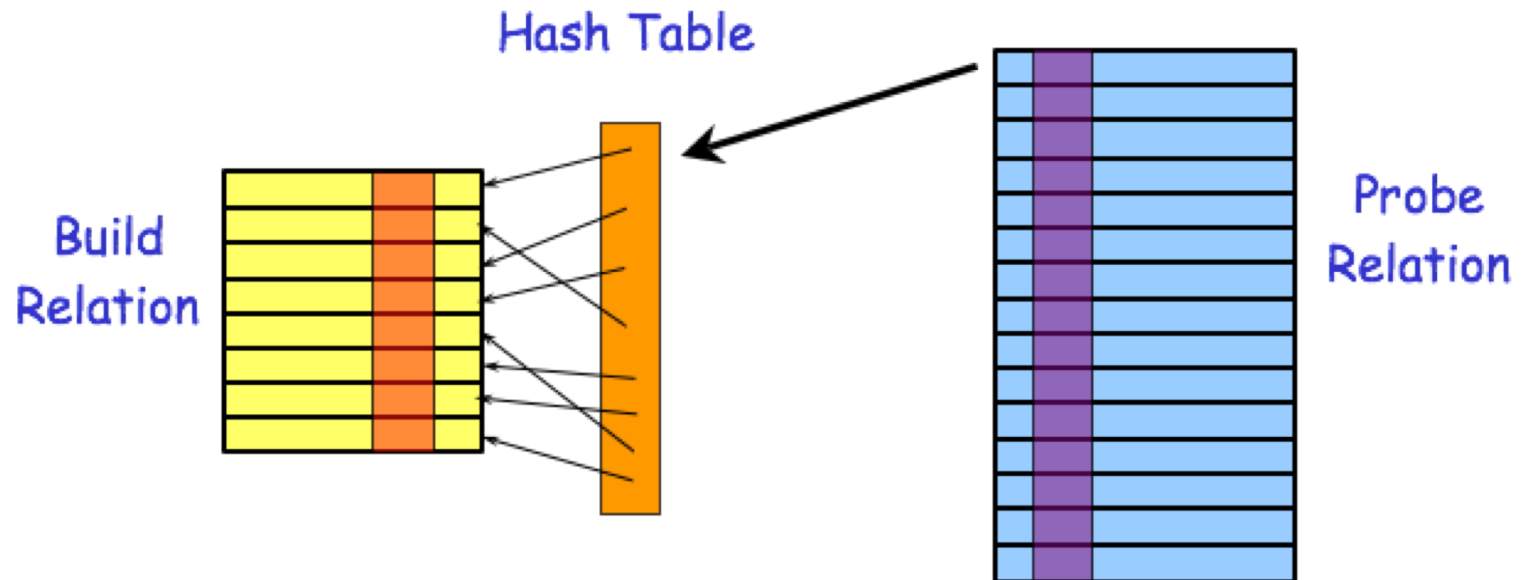
- Message traffic reduced by 7% to 29%
- Relaxed memory consistency → write latency already hidden

Prefetching for Databases

- Hash Join
- Prefetching + SIMD in full queries

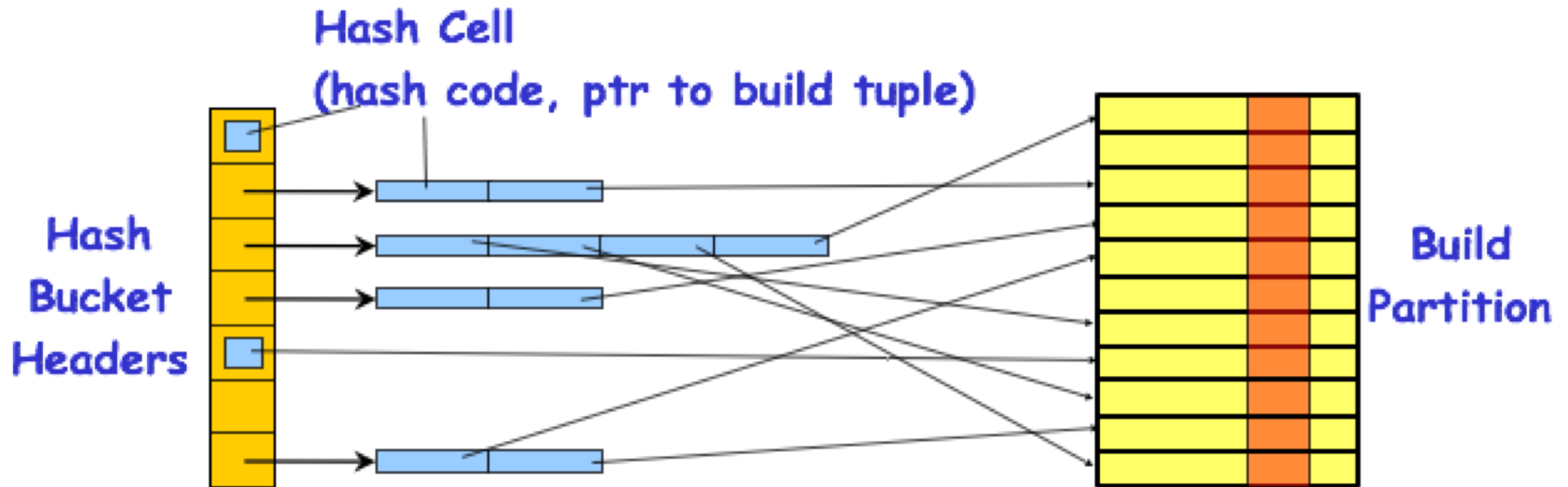
Simple Hash Join

- Build a hash table to index all tuples of the smaller relation
- Probe this hash table using all tuples of the larger relation



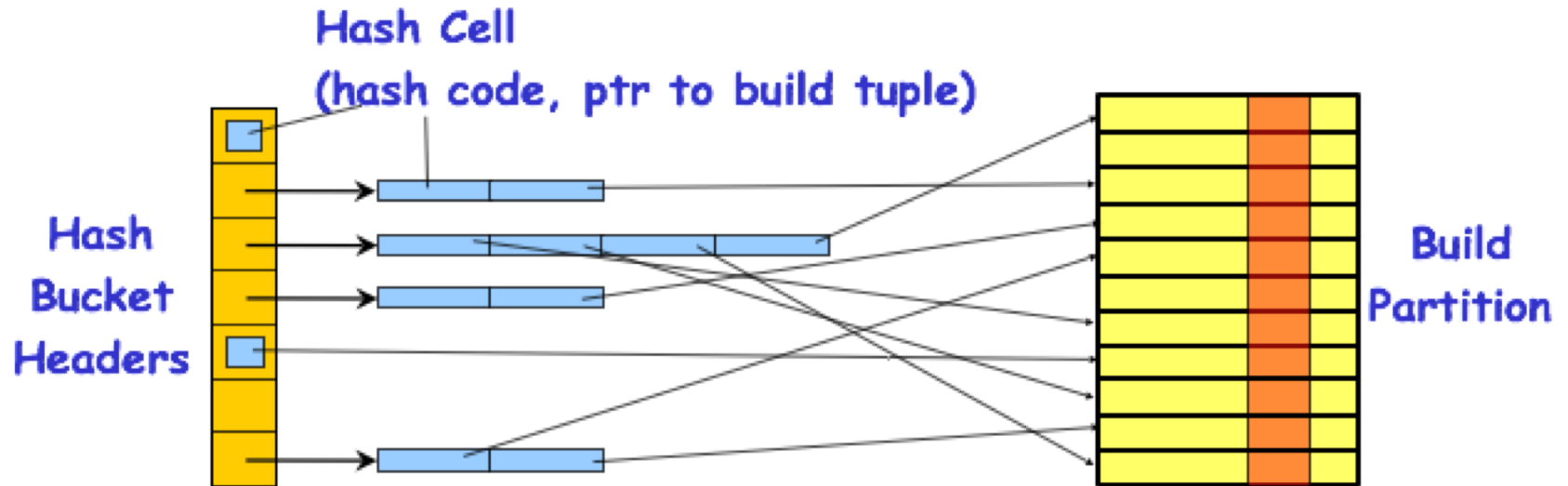
- Random access patterns: little spatial or temporal locality

Challenges



- [Naïve approach](#): prefetch within the processing of a single tuple
 - e.g., prefetch within a single hash table visit
- **Does not work!**
 - dependencies essentially form a critical path
 - addresses would be generated too late for prefetching
 - randomness makes prediction almost impossible

A Simplified Probing Algorithm

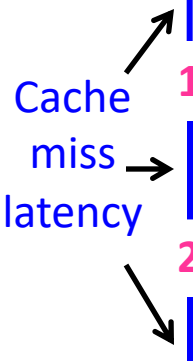


```
foreach tuple in probe partition {  
    compute hash bucket number;  
    visit the hash bucket header;  
    visit the hash cell array;  
    visit the matching build tuple to  
    compare keys and produce output tuple;  
}
```

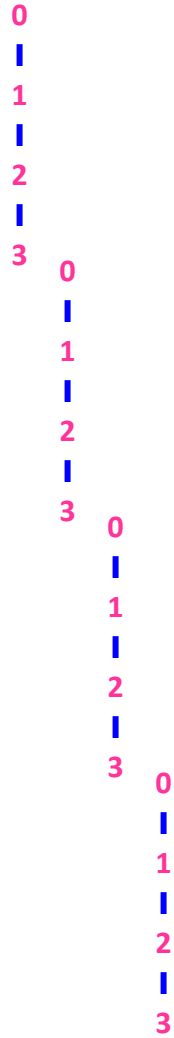
An Intuitive Way to Represent the Algorithm

```
foreach tuple in probe partition {  
    compute hash bucket number;  
  
    visit the hash bucket header;  
  
    visit the hash cell array;  
  
    visit the matching build tuple to  
    compare keys and produce output tuple;  
}
```

Cache
miss
latency



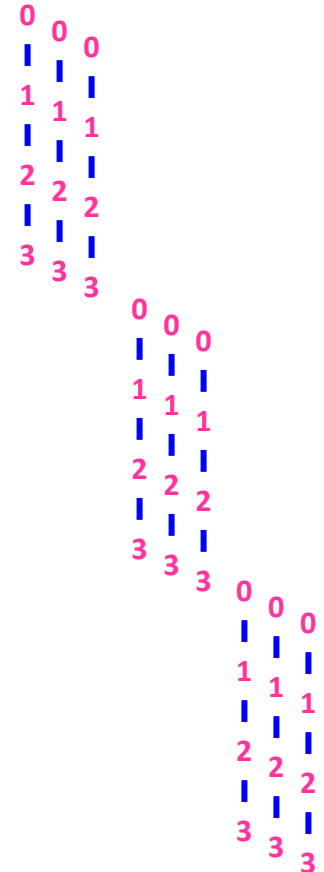
A vertical blue line with indices 0, 1, 2, and 3. Arrows point from the text 'Cache miss latency' to each of these indices.



A vertical blue line with indices 0, 1, 2, and 3 repeated four times, representing a prefetching pattern.

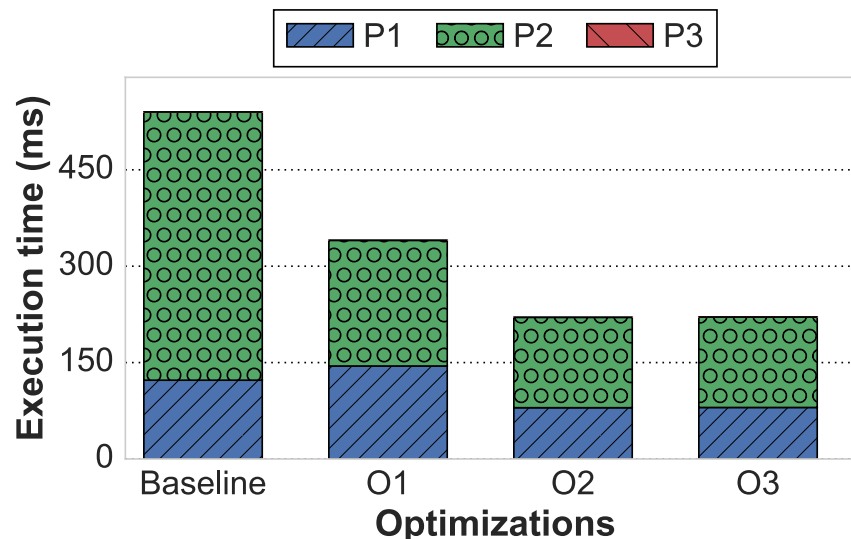
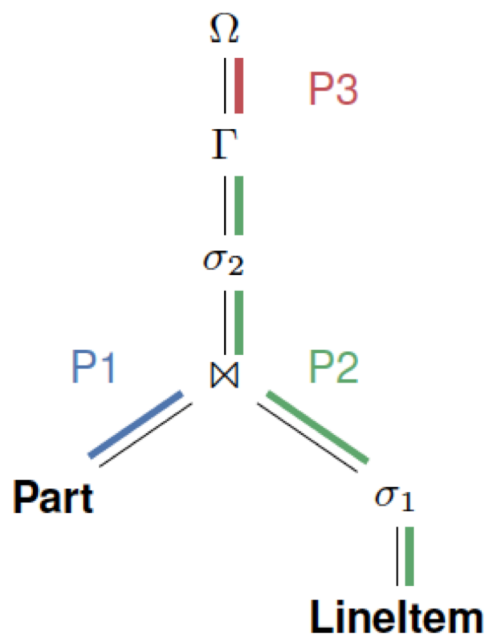
Group Prefetching

```
foreach group of tuples in probe partition {  
  foreach tuple in the group {  
    compute hash bucket number;  
    prefetch the target bucket header;  
  }  
  foreach tuple in the group {  
    visit the hash bucket header;  
    prefetch the hash cell array;  
  }  
  foreach tuple in the group {  
    visit the hash cell array;  
    prefetch the matching build tuple;  
  }  
  foreach tuple in the group {  
    visit the matching build tuple to  
    compare keys and produce output tuple;  
  }  
}
```



Applying Prefetching & SIMD to Queries: e.g., TPC-C Q19

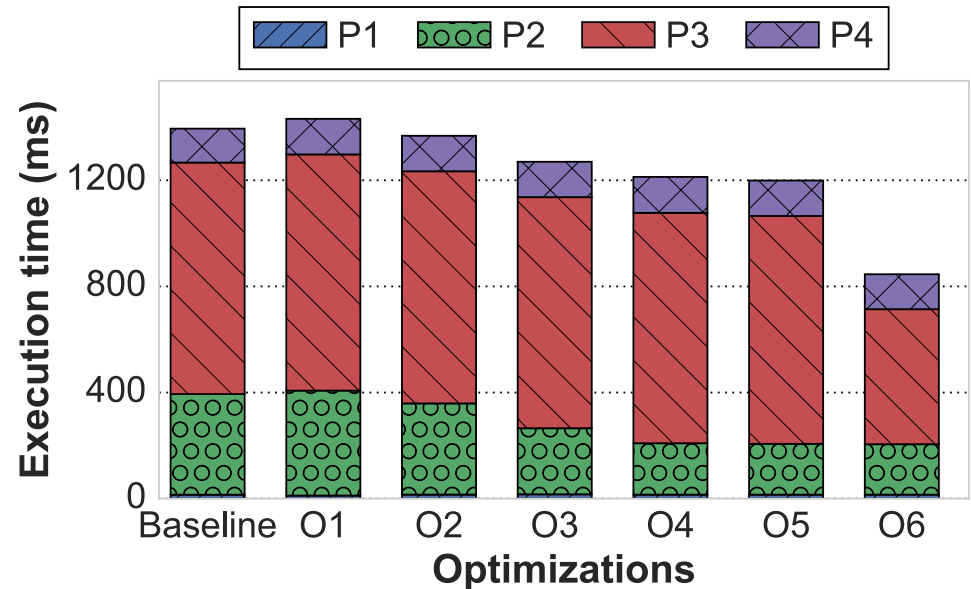
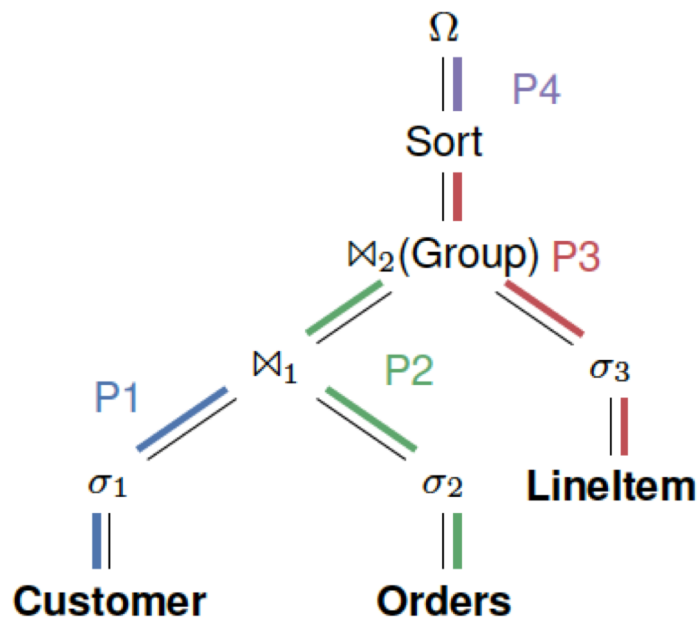
```
SELECT SUM(...) AS revenue
FROM LineItem JOIN Part ON l_partkey = p_partkey
WHERE (CLAUSE1) OR (CLAUSE2) OR (CLAUSE3)
```



O1	Modification: P2 \Rightarrow (LineItem $\rightarrow \sigma_1 \rightarrow \Xi_1 \rightarrow \Join \rightarrow \sigma_2 \rightarrow \Gamma$) Description: Apply SIMD to predicate σ_1 (LineItem).
O2	Modification: — Description: Use Ξ_1 to prefetch buckets during probe of \Join .
O3	Modification: P2 \Rightarrow (LineItem $\rightarrow \sigma_1 \rightarrow \Xi_1 \rightarrow \Join \rightarrow \Xi_2 \rightarrow \sigma_2 \rightarrow \Xi_3 \rightarrow \Gamma$) Description: Insert staging points between every pair of operators.

- SIMD and prefetching are complementary: over 2X speedup

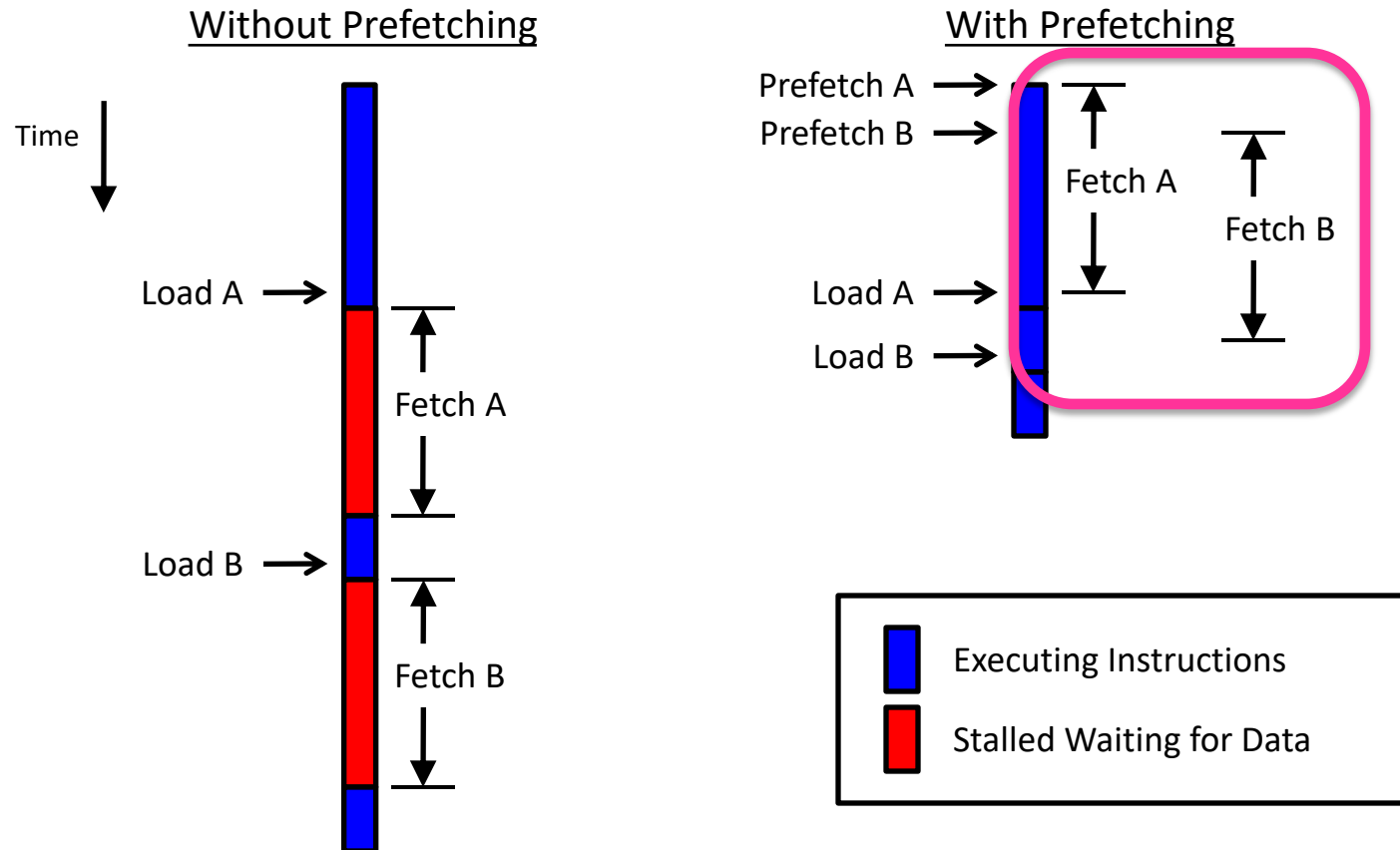
A More Complex Query from TPC-C: Q3



O1	Modification: $P1 \Rightarrow (\text{Customer} \rightarrow \sigma_1 \rightarrow \Xi_1 \rightarrow \bowtie_1)$ Description: Apply SIMD to predicate σ_1 (Customer).
O2	Modification: $P2 \Rightarrow (\text{Orders} \rightarrow \sigma_2 \rightarrow \Xi_2 \rightarrow \bowtie_1 \rightarrow \bowtie_2)$ Description: Apply SIMD to predicate σ_2 (Orders).
O3	Modification: — Description: Use Ξ_2 to prefetch buckets during \bowtie_1 probe.
O4	Modification: $P2 \Rightarrow (\text{Orders} \rightarrow \sigma_2 \rightarrow \Xi_2 \rightarrow \bowtie_1 \rightarrow \Xi_3 \rightarrow \bowtie_2)$ Description: Use Ξ_3 to prefetch buckets during build of \bowtie_2 .
O5	Modification: $P3 \Rightarrow (\text{LineItem} \rightarrow \sigma_3 \rightarrow \Xi_4 \rightarrow \bowtie_2 \rightarrow \text{Sort})$ Description: Apply SIMD to predicate σ_3 .
O6	Modification: — Description: Use Ξ_4 to prefetch buckets for \bowtie_2 probe.

- SIMD + Prefetching \rightarrow large improvement

Prefetching Only Works if there is Sufficient Memory Bandwidth

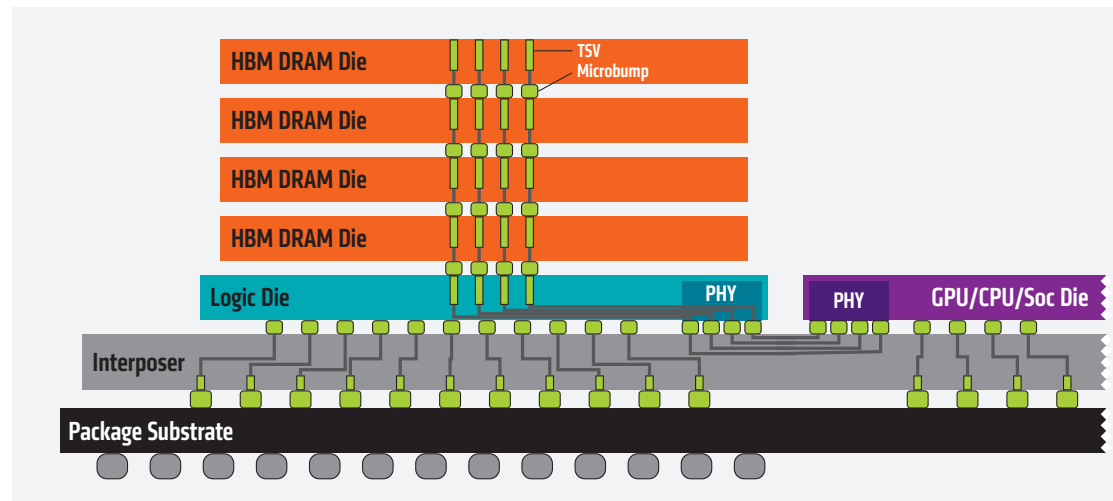


- If you are already bandwidth-limited, then prefetching cannot help

How Can We Provide Sufficient Memory Bandwidth?

Recent enabling technology: 3D stacking of DRAM chips

- DRAMs connected via through-silicon-vias (TSVs) that run through the chips
 - TSVs provide highly parallel connection between logic layer and DRAMs
- Base layer of stack “logic layer” is memory controller, manages requests from processor
- Silicon “interposer” serves as high-BW interconnect between DRAM stack and processor



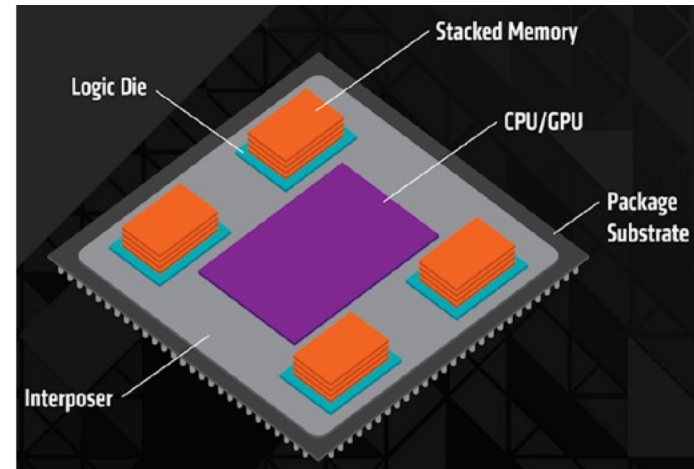
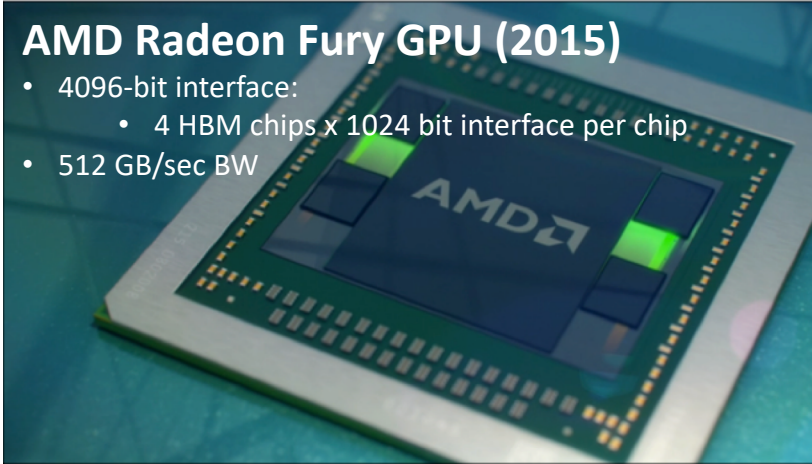
Technologies:

- Micron/Intel’s Hybrid Memory Cube (HMC)
- AMD’s High-Bandwidth Memory (HBM): 1024 bit interface to stack

GPUs Have Been Adopting HBM Technologies

AMD Radeon Fury GPU (2015)

- 4096-bit interface:
 - 4 HBM chips x 1024 bit interface per chip
- 512 GB/sec BW



NVIDIA P100 GPU (2016)

- 4096-bit interface:
 - 4 HBM2 chips x 1024 bit interface per chip
- 720 GB/sec peak BW
- 4 x 4 GB = 16 GB capacity

- ```
// allocate buffer in MCDRAM ("high bandwidth" memory malloc)
float* foo = hbw_malloc(sizeof(float) * 1024);
```



## Bonus Material: Prefetching for Recursive Data Structures

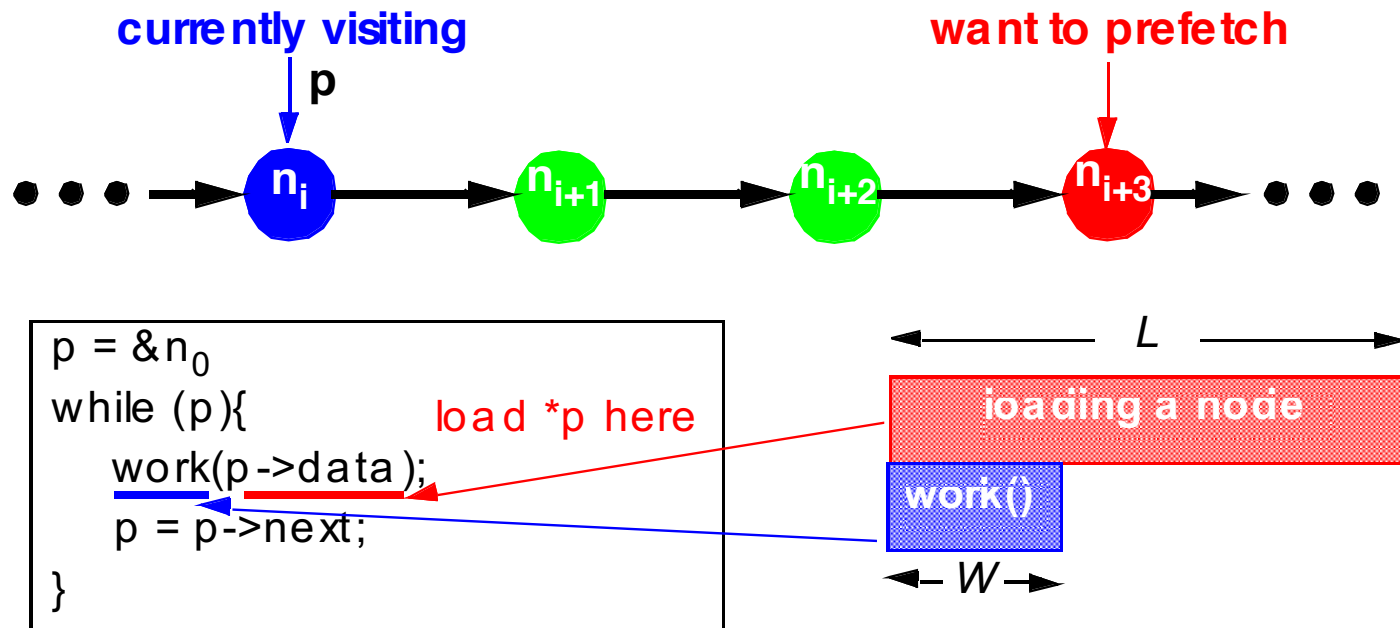
- Examples:
  - linked lists, trees, graphs, ...
- A common method of building large data structures
  - especially in non-numeric programs
- Cache miss behavior is a concern because:
  - large data set with respect to the cache size
  - temporal locality may be poor
  - little spatial locality among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

### Goal:

- Automatic Compiler-Based Prefetching for Recursive Data Structures



# Scheduling Prefetches for Recursive Data Structures

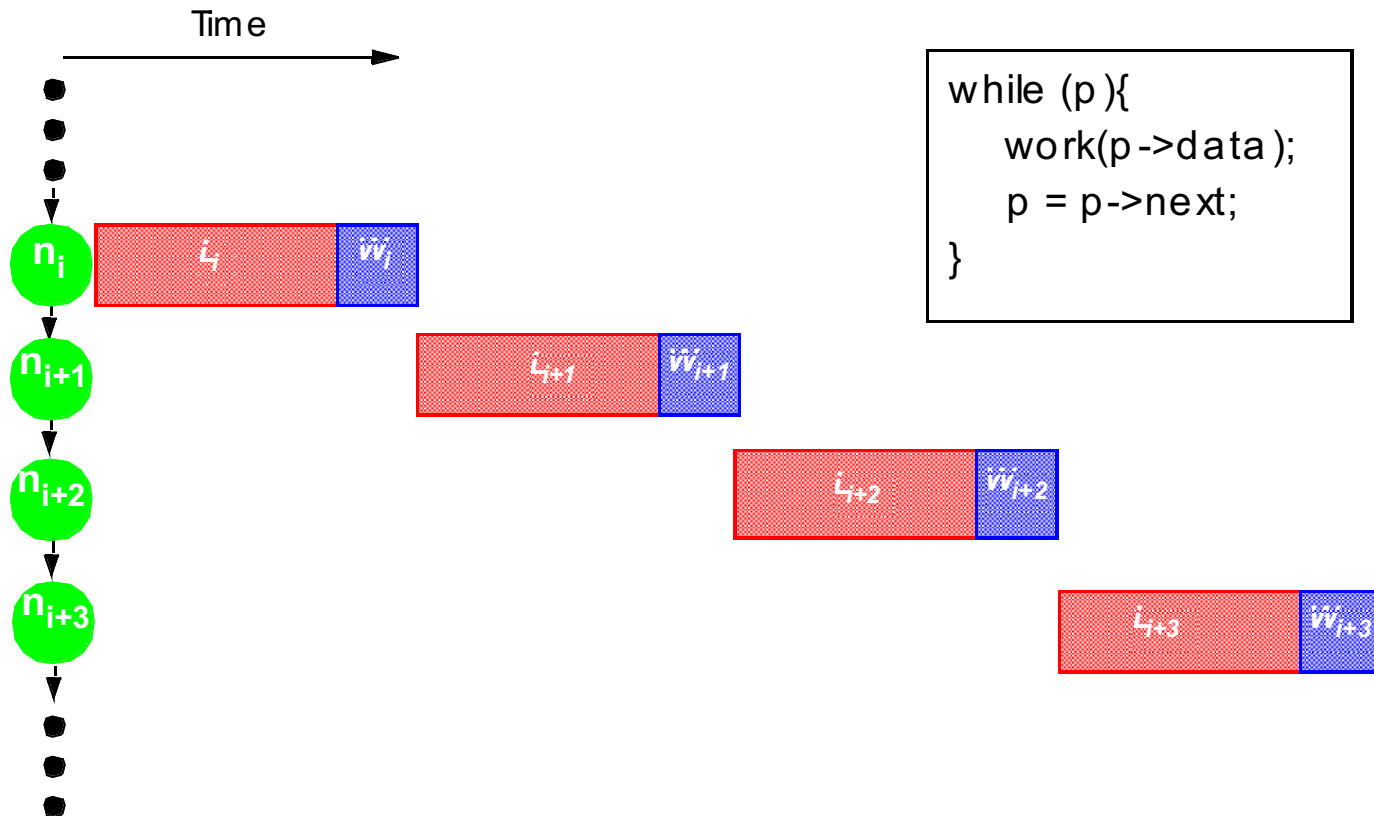


Our Goal: *fully hide latency*

– thus achieving fastest possible computation rate of  $1/W$

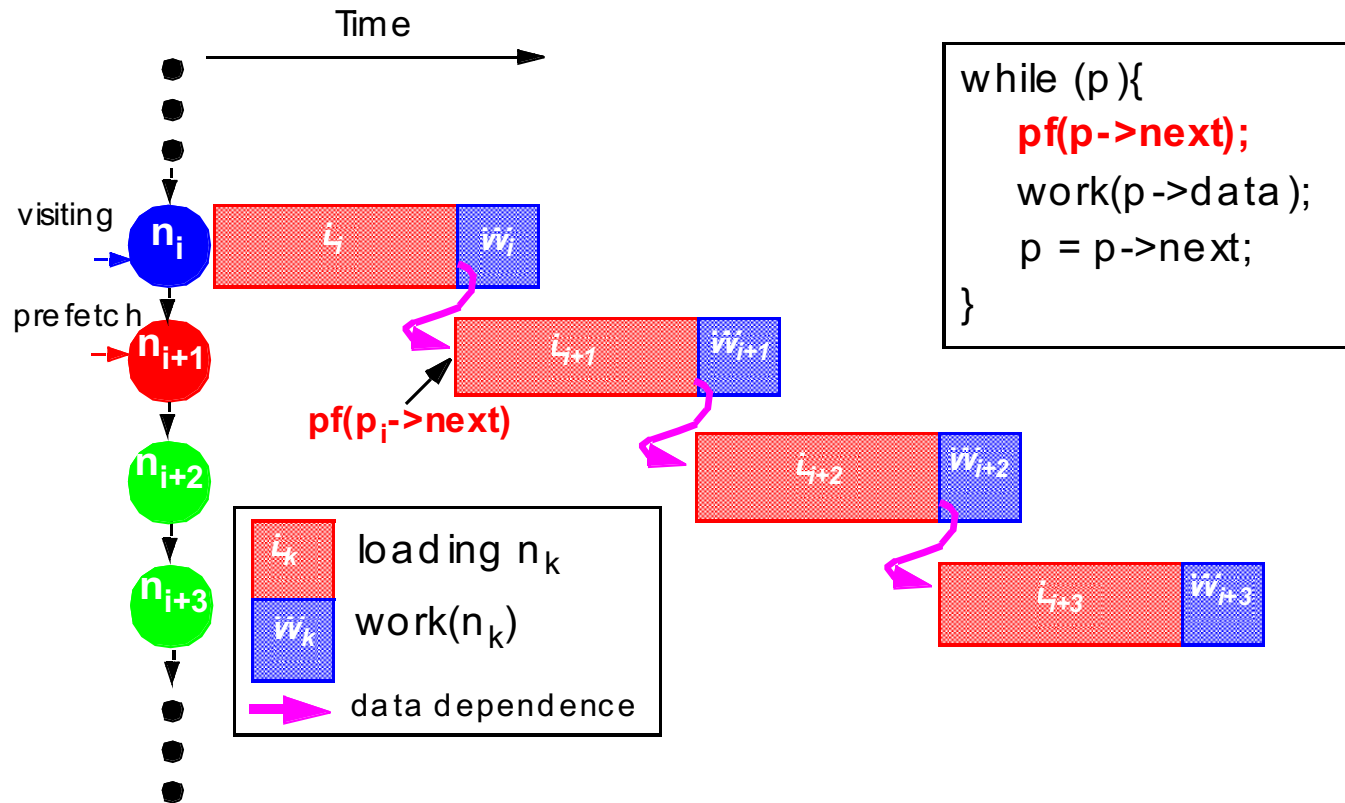
- e.g., if  $L = 3W$ , we must prefetch 3 nodes ahead to achieve this

## Performance without Prefetching



$$\text{computation rate} = 1 / (L+W)$$

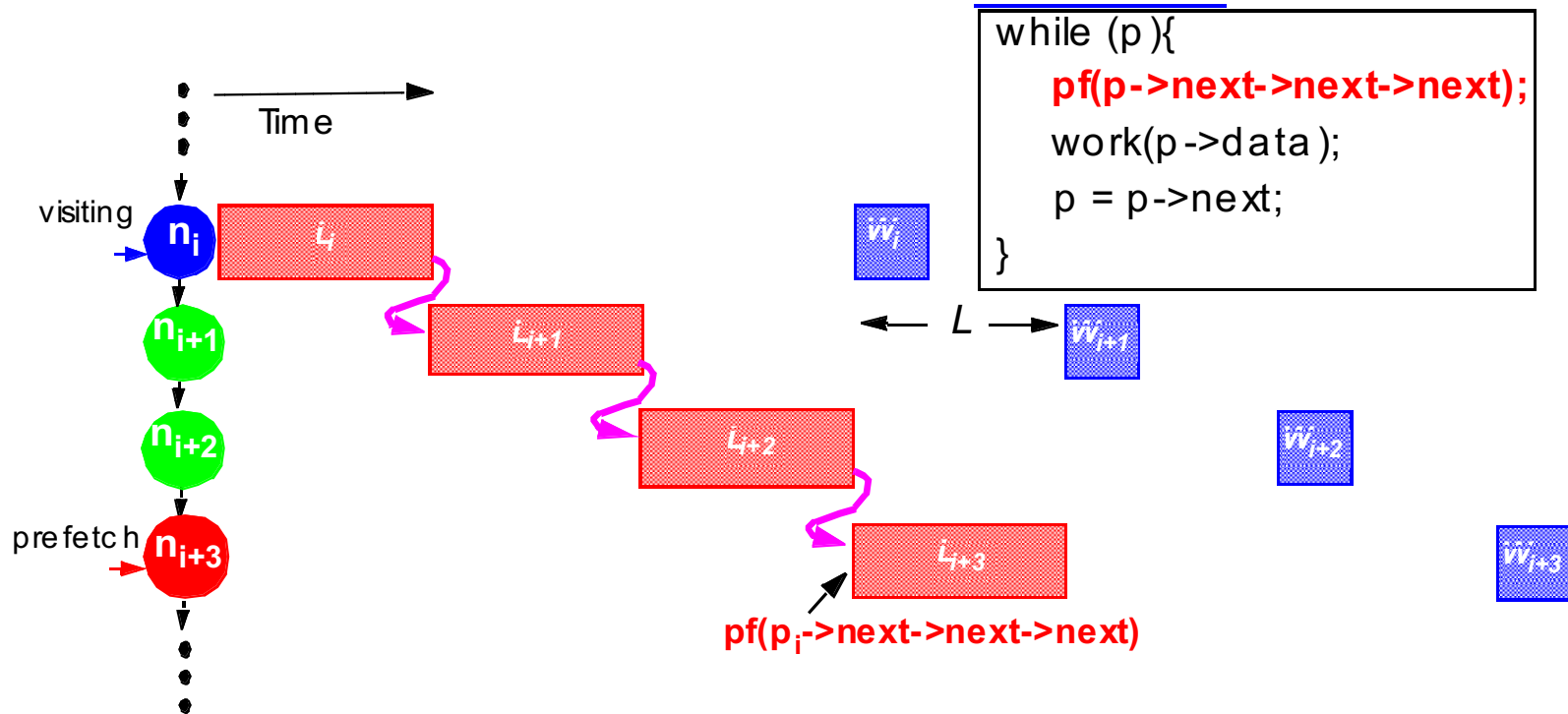
# Prefetching One Node Ahead



- Computation is overlapped with memory accesses

computation rate =  $1/L$

## Prefetching Three Nodes Ahead

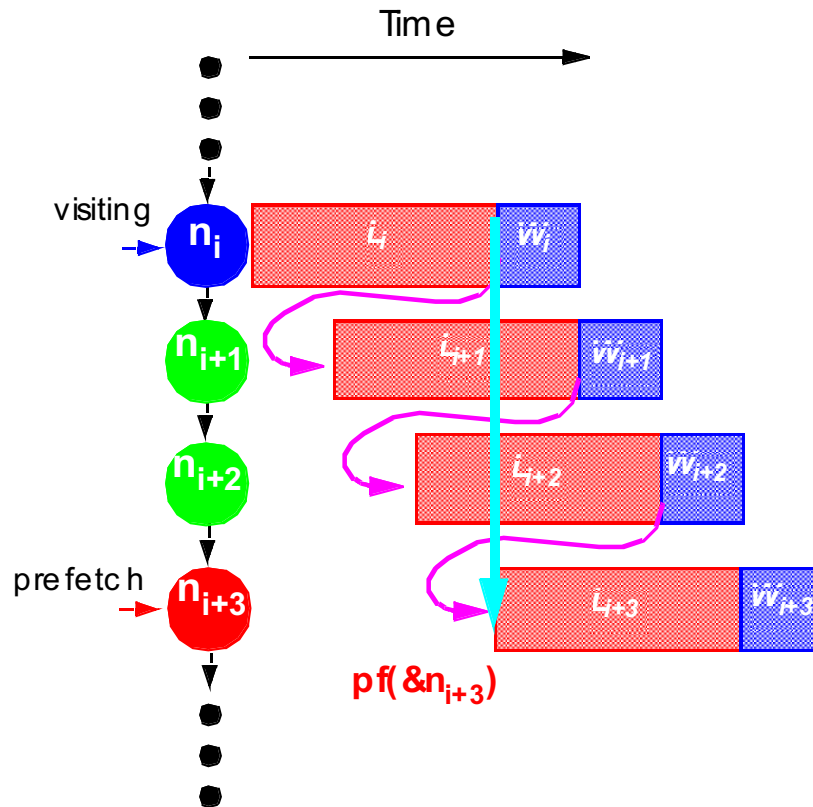


*computation rate does not improve (still =  $1/L$ )!*

### Pointer-Chasing Problem:

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

## Our Goal: Fully Hide Latency



```
while (p){
 pf(&ni+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

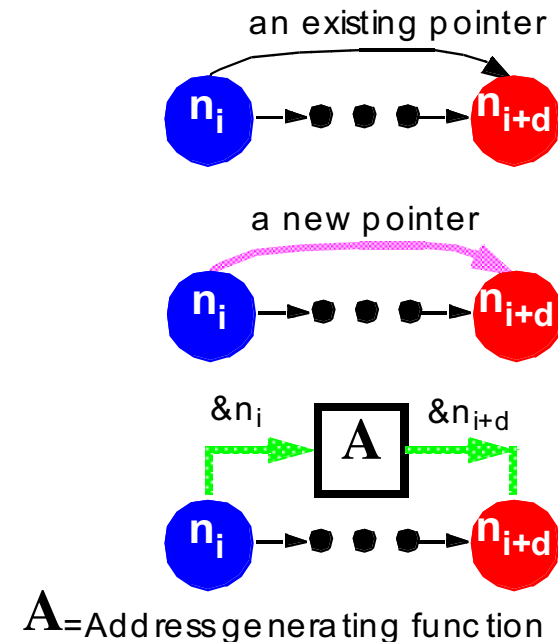
# Overcoming the Pointer-Chasing Problem

Key:

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

Our proposals:

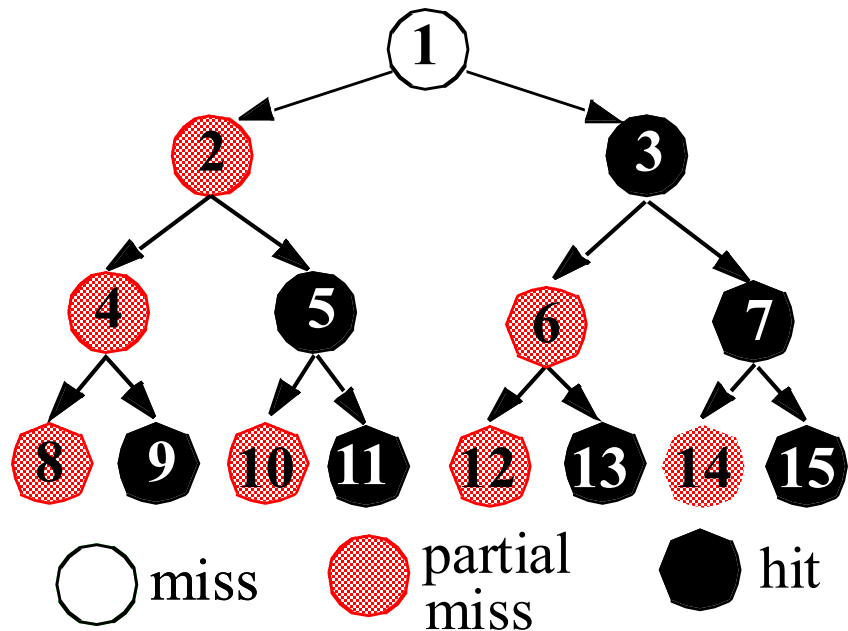
- use *existing* pointer(s) in  $n_i$  to approximate  $\&n_{i+d}$ 
  - Greedy Prefetching
- add *new* pointer(s) to  $n_i$  to approximate  $\&n_{i+d}$ 
  - History-Pointer Prefetching
- compute  $\&n_{i+d}$  *directly* from  $\&n_i$  (no ptr deref)
  - History-Pointer Prefetching



## 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);
 }
}
```

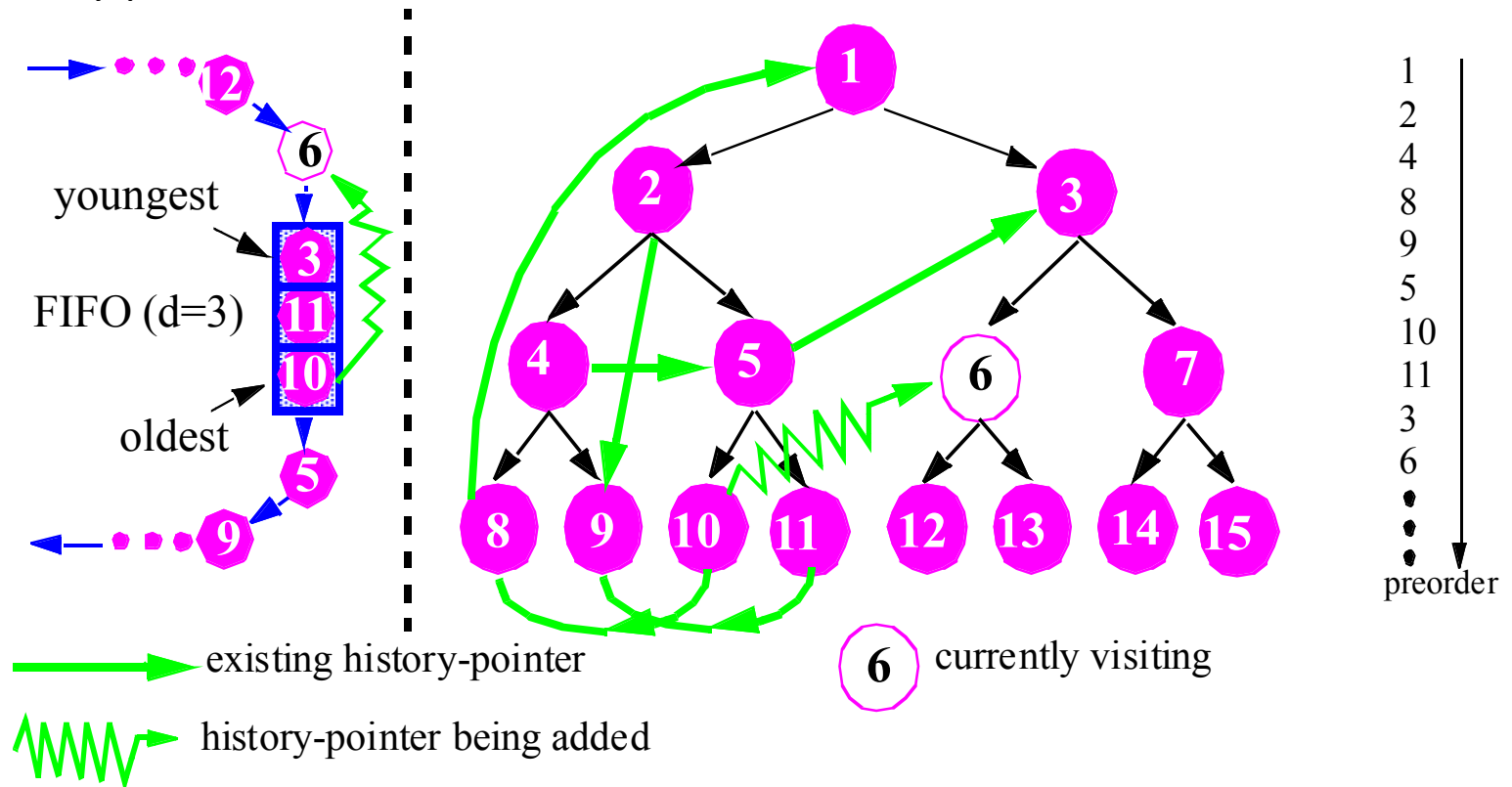


- Reasonably effective in practice
- However, little control over the prefetching distance



# History-Pointer Prefetching

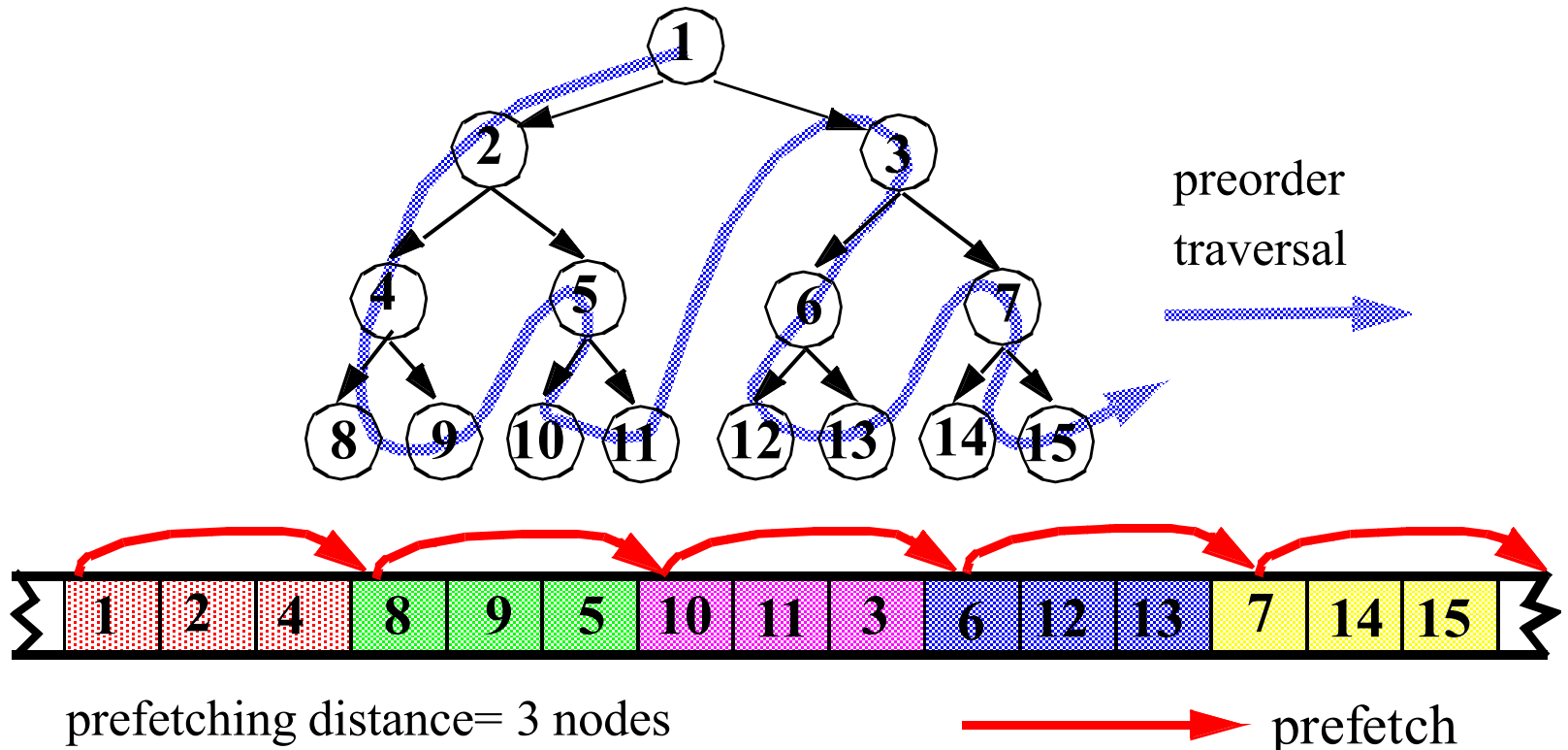
- Add new pointer(s) to each node
  - history-pointers are obtained from some recent traversal



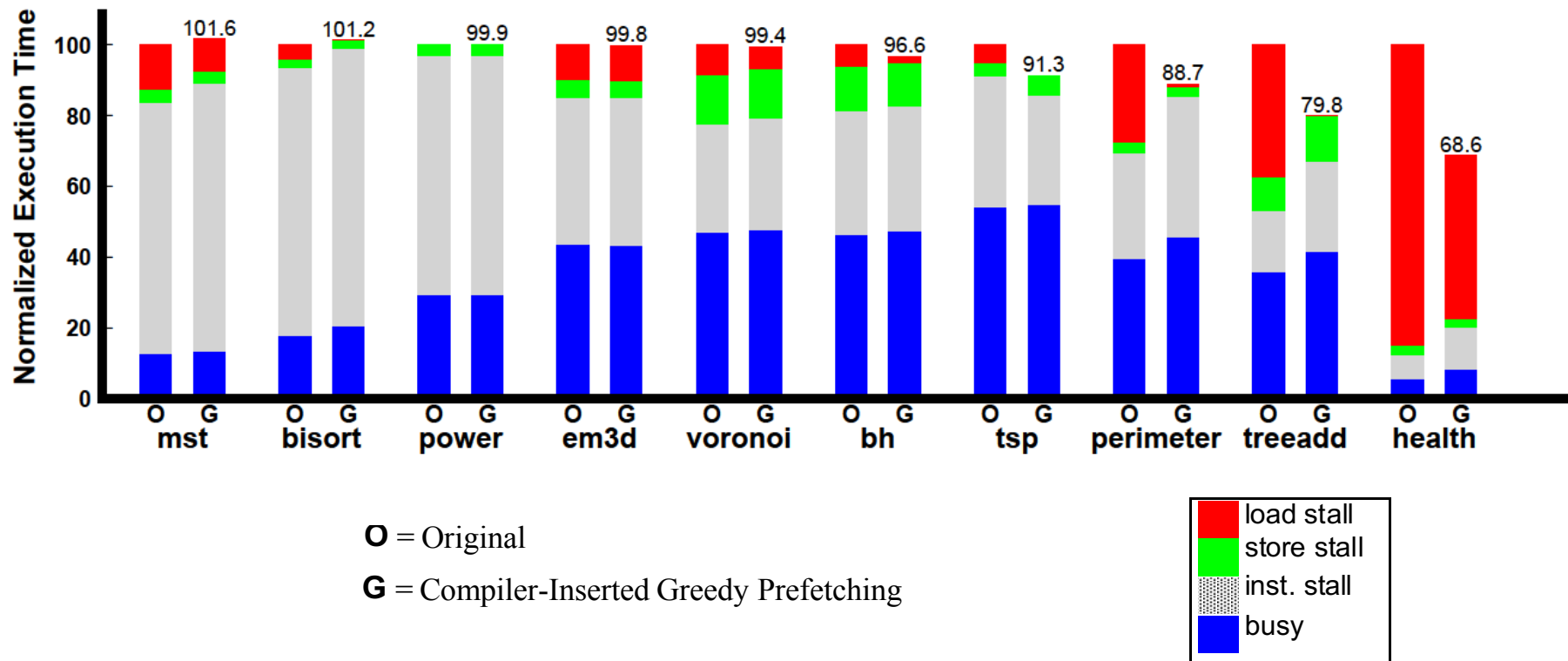
- Trade space & time for better control over prefetching distances

## Data-Linearization Prefetching

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

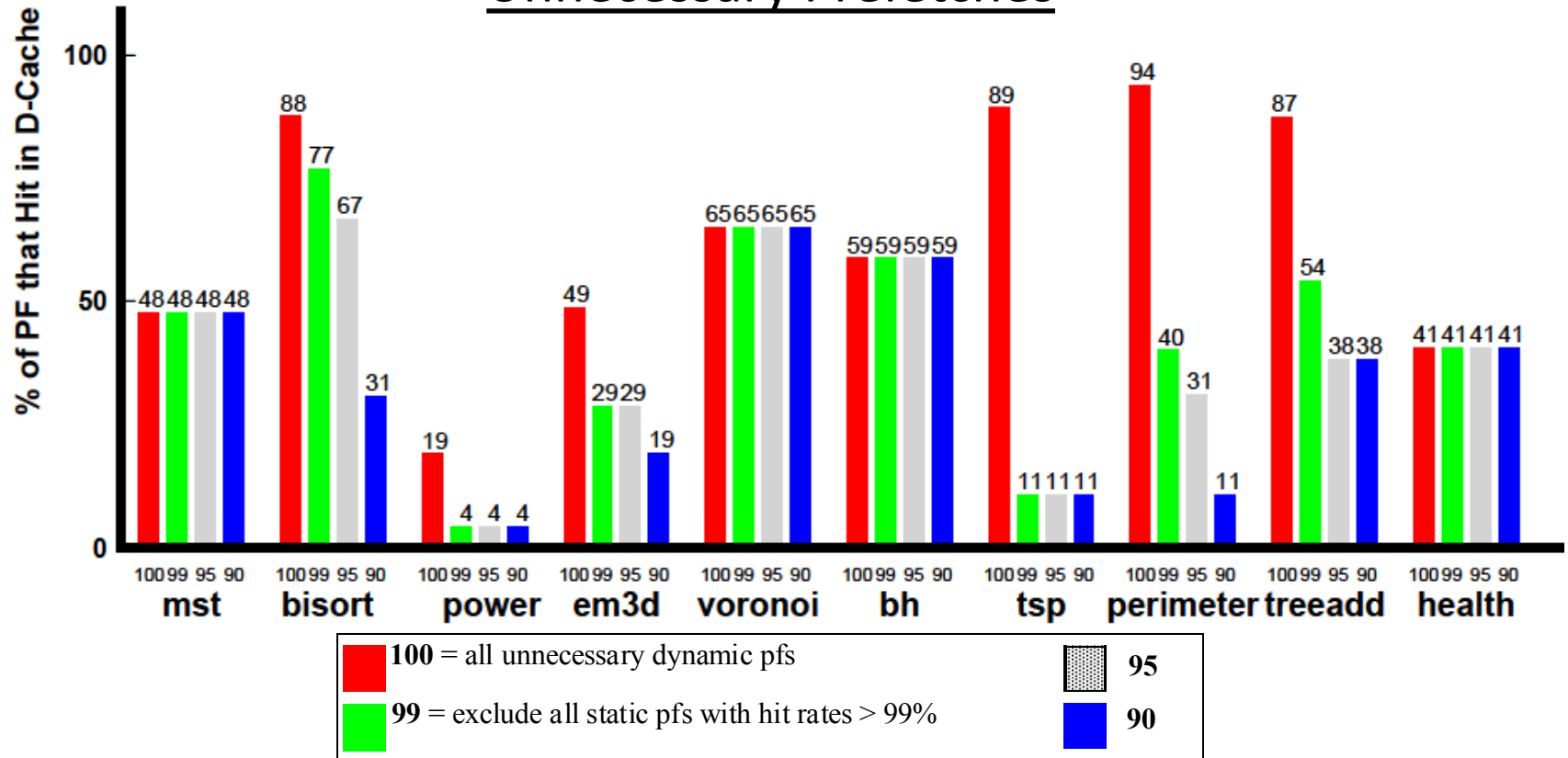


## Performance of Compiler-Inserted Greedy Prefetching



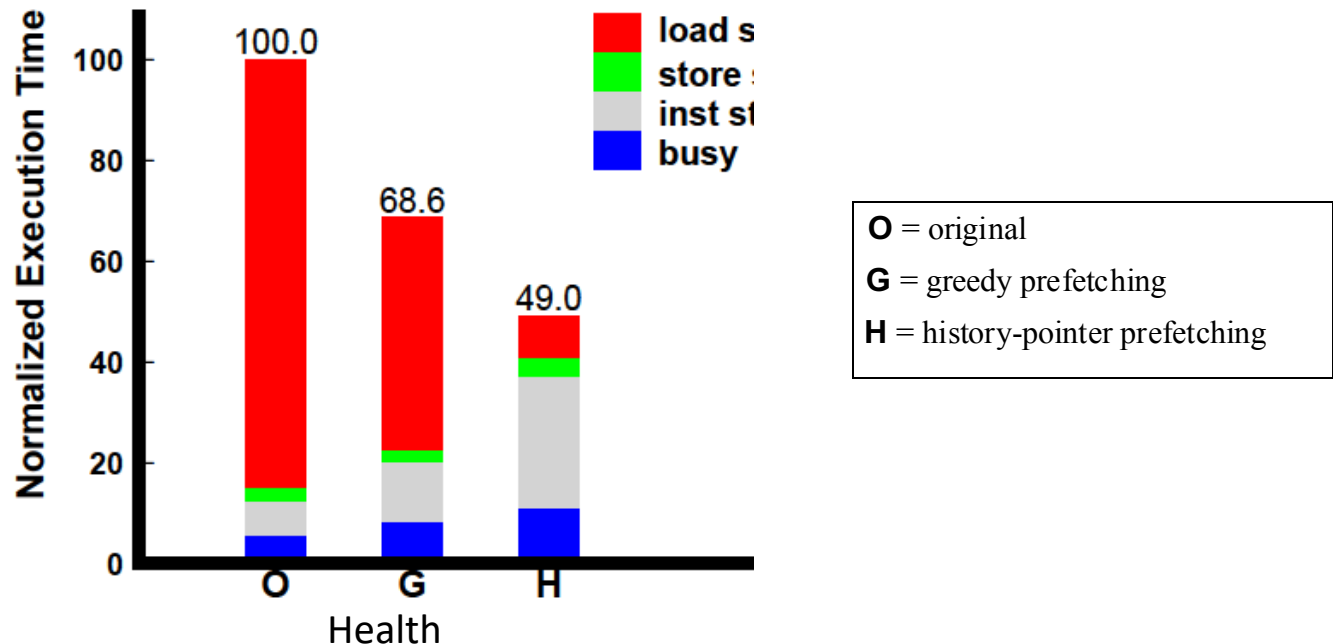
- Eliminates much of the stall time in programs with large load stall penalties
  - half achieve speedups of 4% to 45%

# Unnecessary Prefetches



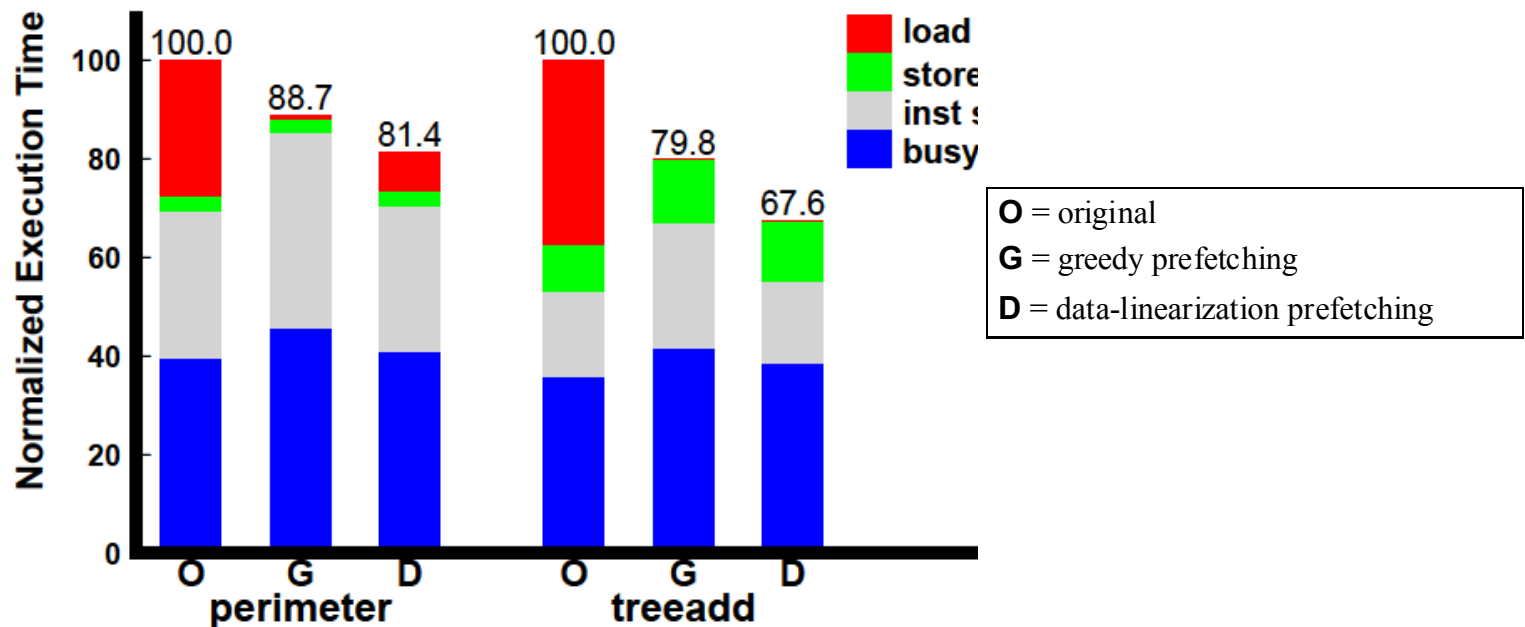
- % dynamic pfs that are unnecessary because the data is in the D-cache
- 4 have >80% unnecessary prefetches
- Could reduce overhead by eliminating static pfs that have high hit rates

## 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

- 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