Optimizing the Cache Performance of Non-Numeric Applications

Chi-Keung Luk

Department of Computer Science

University of Toronto

&

Computer Science Department

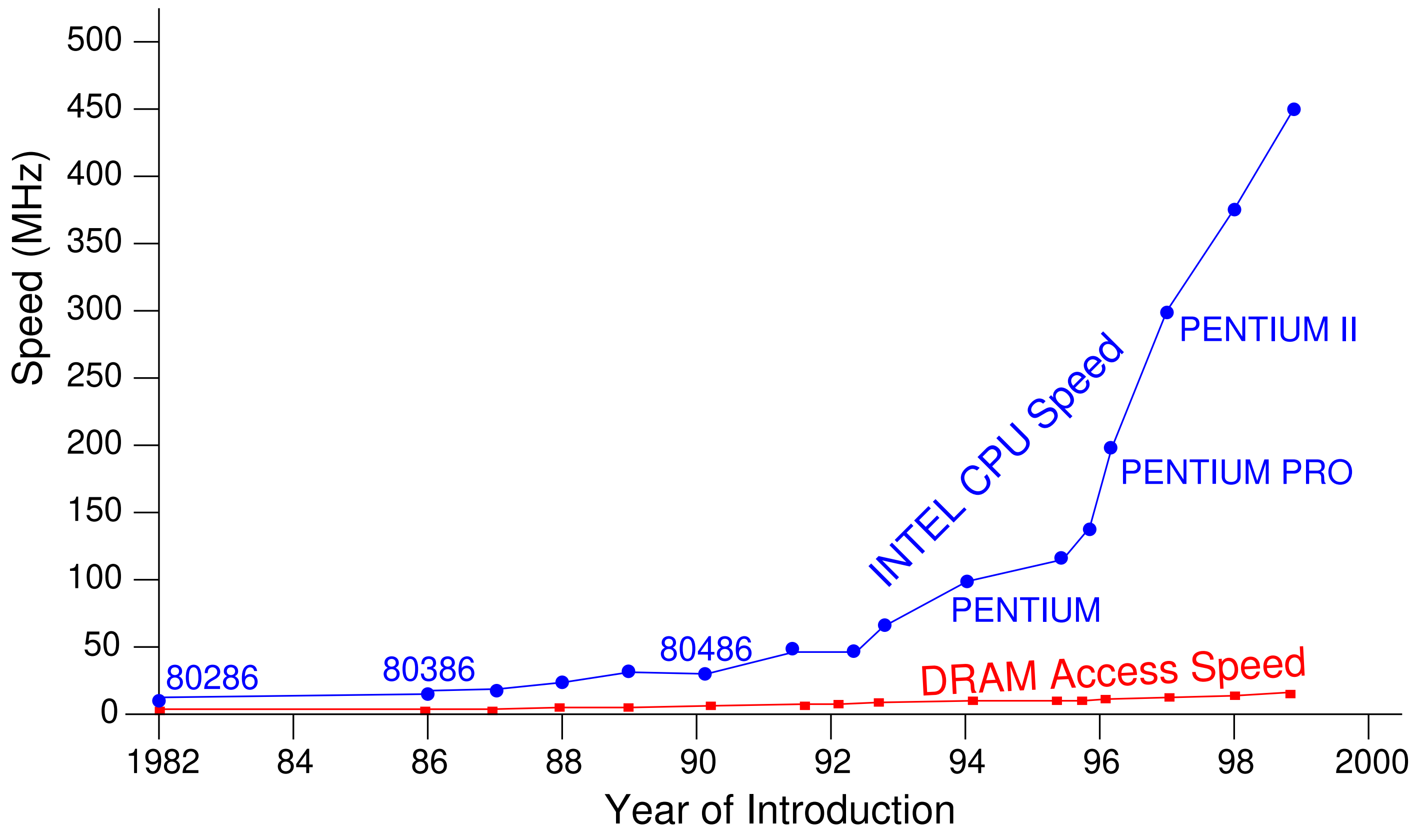
Carnegie Mellon University

luk@cs.cmu.edu

http://www.cs.cmu.edu/~luk

The Memory Latency Problem

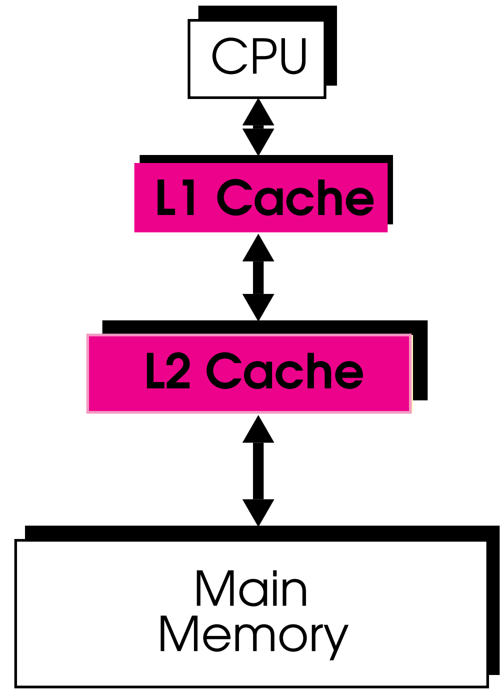
- Memory subsystem supplies both instructions and data
- ↑ processor speed >> ↑ memory speed



☞ Memory latency greatly limits overall performance

Caches

- **Caches are an important first step**
  - storing frequently used data into faster (but smaller) memory
  - relying on locality of reference



- **But *not* a complete solution**

Numeric Codes vs. Non-Numeric Codes

	Numeric	Non-Numeric

Examples	for (i = 0; i < 100; i++) {
	  for (k = 0; k < 100; k++) {
	    A(i)(k) = 3.1416 * B(k)(i);
	  }
	}
		walk (treeNode* p) {
		  if (continue(p)){
		    walk(p->left);
		    walk(p->right);
		  }else
		    compute(p->data);
		}

Characteristics	floating point operations, regular loop nests, multidimensional arrays	integer operations, irregular control flow, pointer-linked objects
Application Domains	Scientific, Engineering	Commercial

☞ **Non-numeric codes are far more common but also difficult to optimize**

Cache Performance on Non-Numeric Codes

- **Both instruction and data cache miss penalties can be significant**
  - especially true in commercial applications (Barroso *et al*, 1998):
    - >75% exec. of an Oracle app. stalled evenly on I-misses & D-misses

- **Lack of research and techniques to hide latency in non-numeric codes**
  - techniques that succeed in numeric codes are of little help

☞ **need latency-hiding techniques that target non-numeric codes**

Our Latency-Hiding Strategy

Data Access Latency: (primary focus is on *pointer-based data structures*)

- tolerate latency through software-controlled prefetching
  - ☞ techniques to predict future data addresses
    - ☞ **compiler-based prefetching for recursive data structures (ASPLOS '96, IEEE TC '99)**
- minimize overheads of software-based latency tolerance methods
  - ☞ techniques to predict when cache misses happen
    - ☞ **correlation-based miss prediction (MICRO '97, IEEE TC)**
- reduce latency by relocating data dynamically
  - ☞ techniques to guarantee the safety of data relocation
    - ☞ **memory forwarding (ISCA '99)**

Instruction Access Latency:

- tolerate latency through instruction prefetching
  - ☞ techniques to schedule prefetches far enough in advance
    - ☞ cooperative instruction prefetching **(MICRO '98)**

Overview

- ✓ Coping with Memory Latency
- ☞ Improving Data Cache Performance
  - ☞ Compiler-Based Prefetching for Recursive Data Structures
  - Memory Forwarding
- Improving Instruction Cache Performance
  - Cooperative Instruction Prefetching
- Conclusions

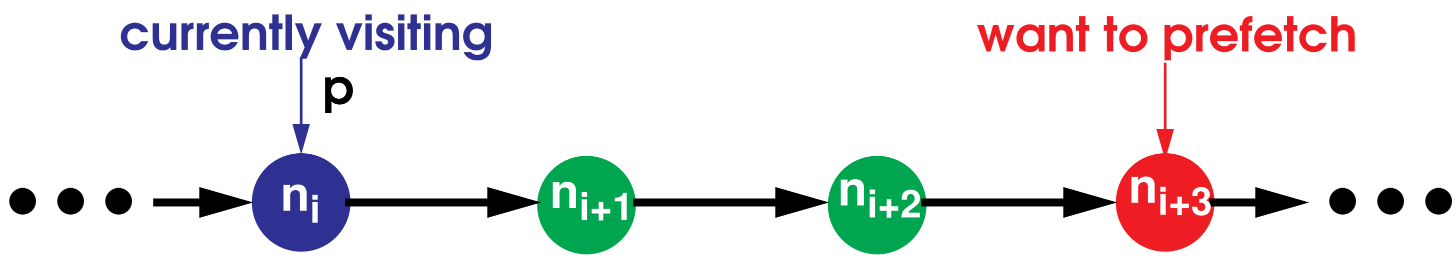
Recursive Data Structures

- **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

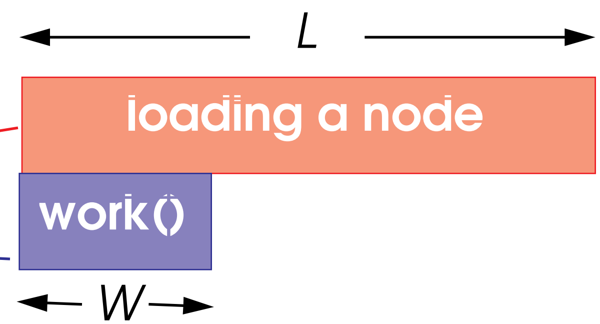
Goal:

☞ **Automatic compiler-based prefetching for recursive data structures**

Scheduling Prefetches for Recursive Data Structures




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

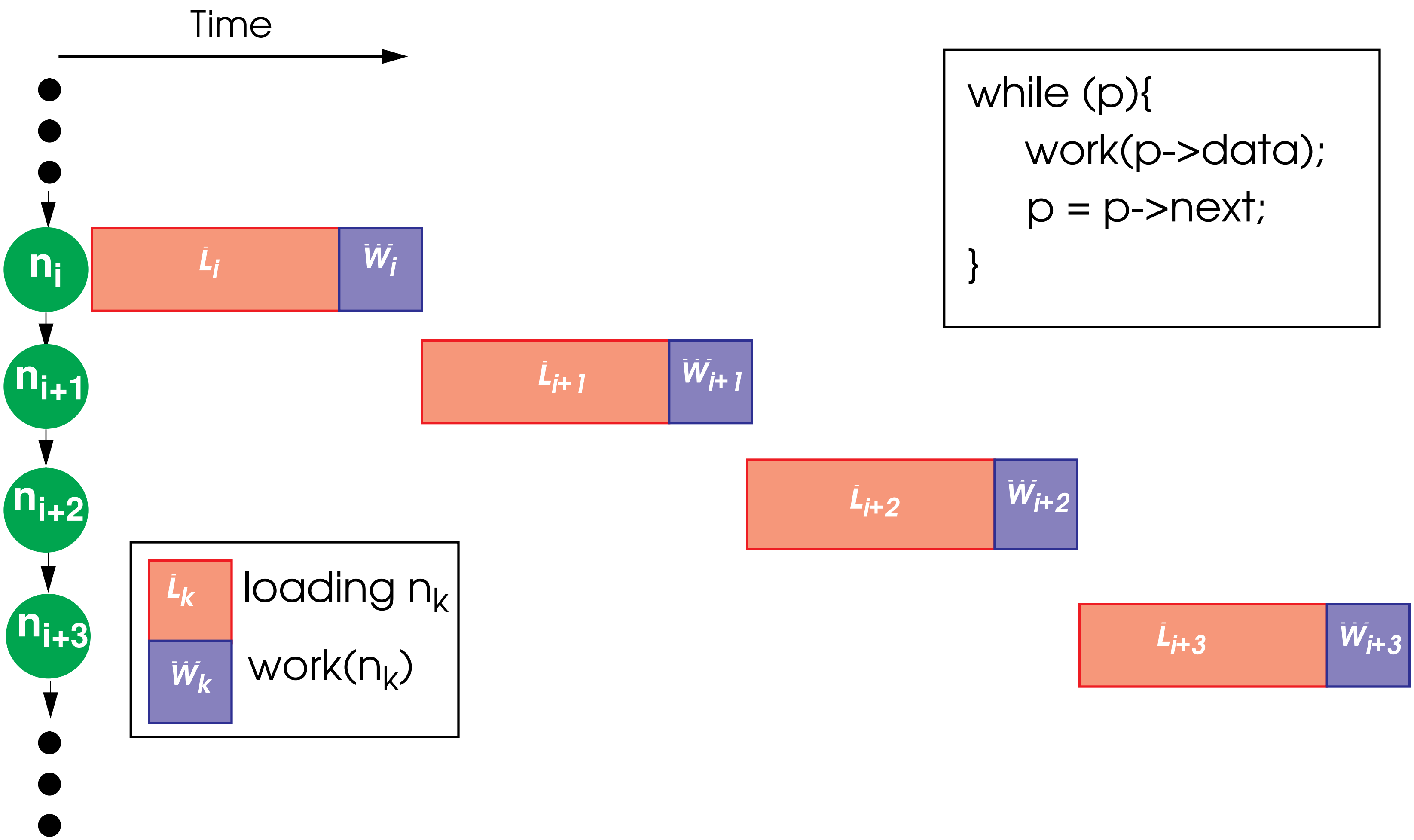


Our Goal: *fully hide latency*

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

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

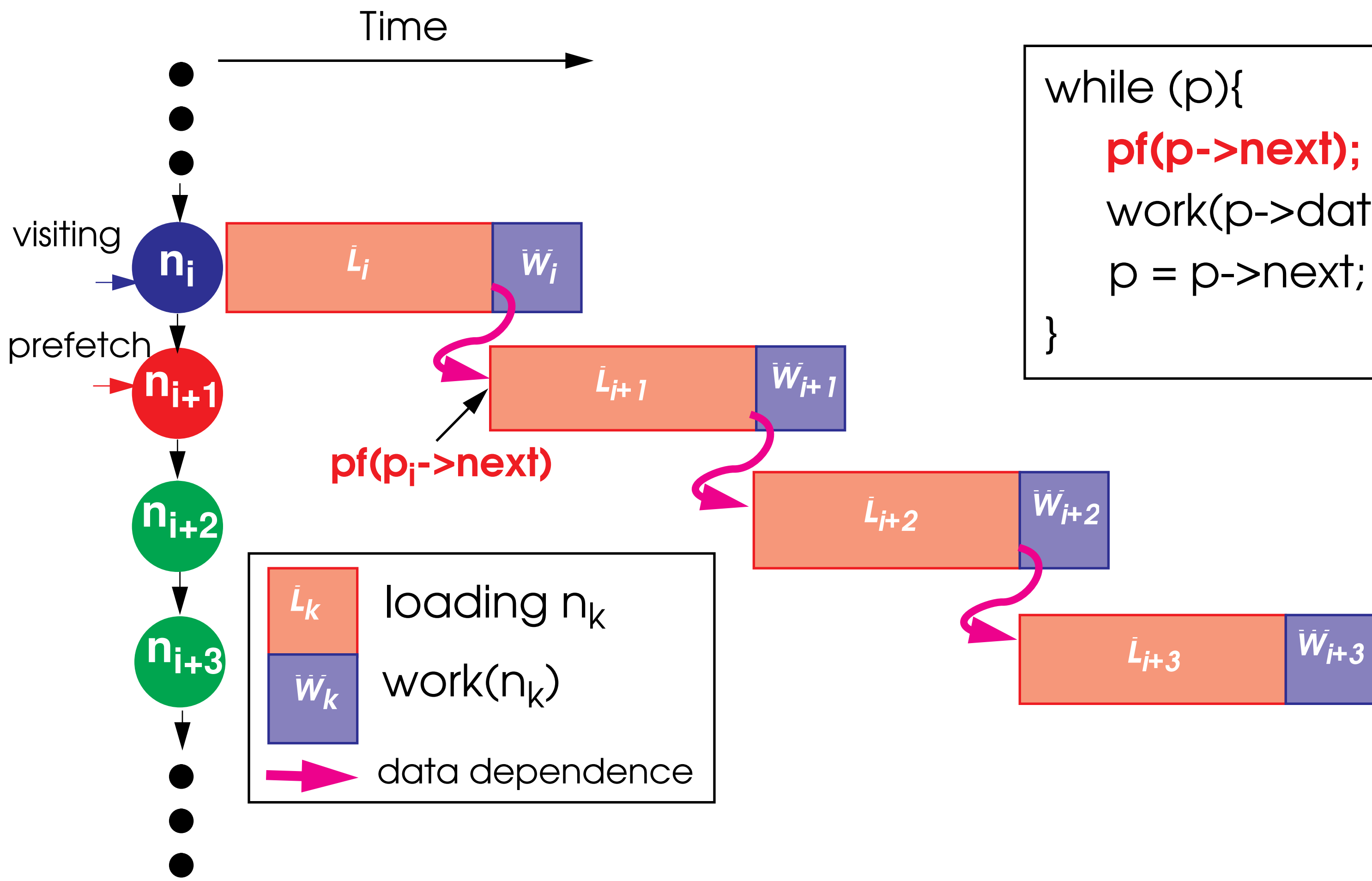
Performance without Prefetching




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

☞ **computation rate = 1/(L+W)**

Prefetching One Node Ahead



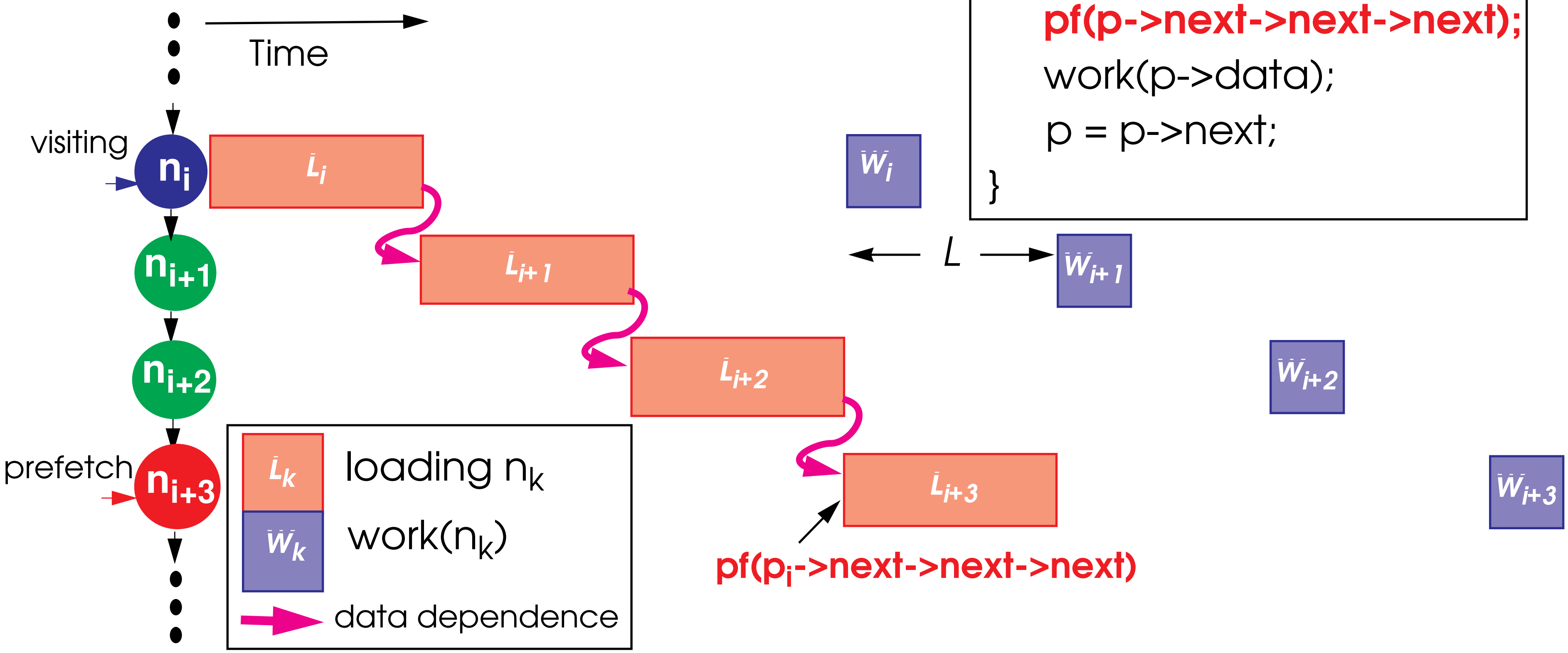

```
while (p){
    pf(p->next);
    work(p->data);
    p = p->next;
}
```

- 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;
}
```

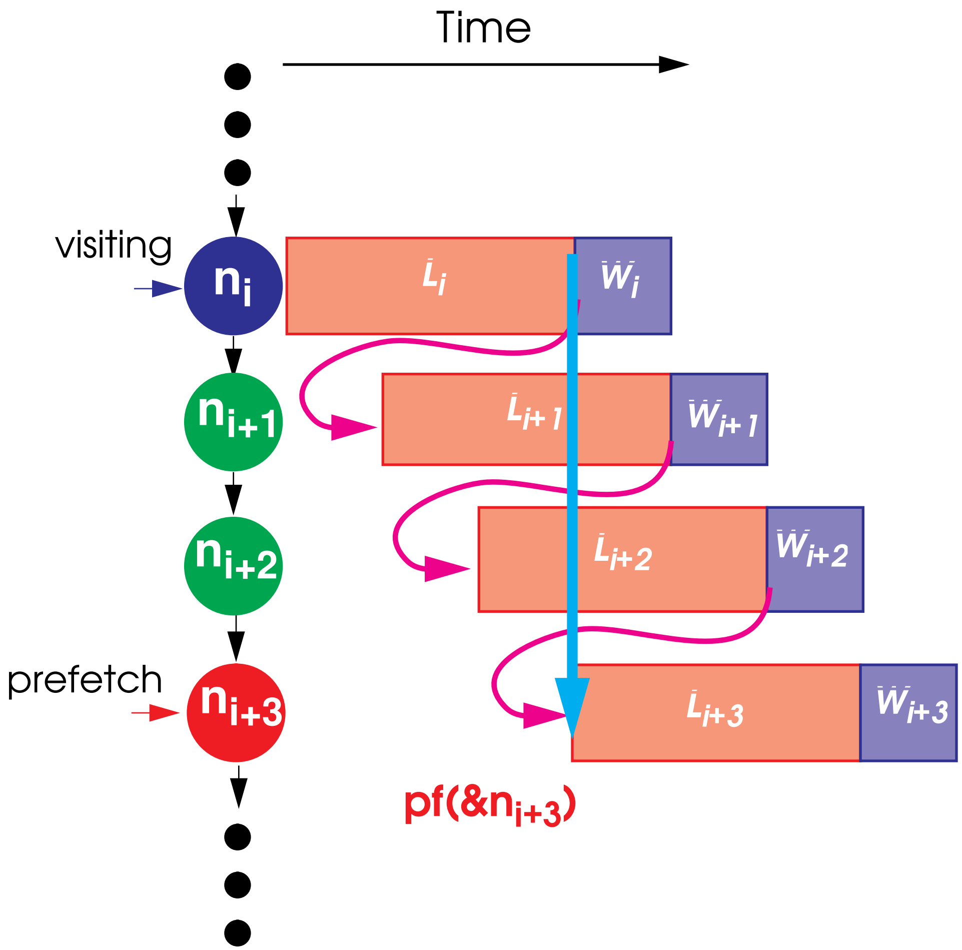


☞ ***computation rate does not improve* (still = 1/*L*)!**

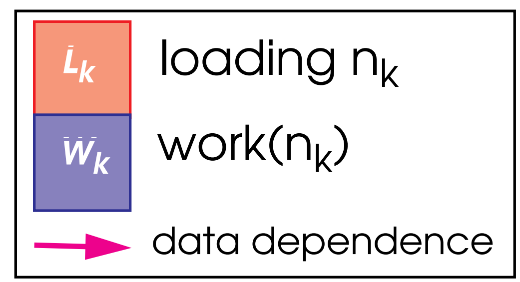
Pointer-Chasing Problem:

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 $1/W$**

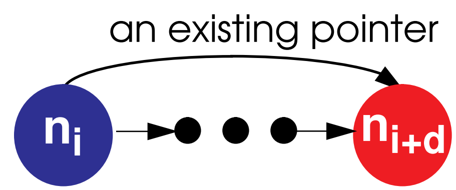
Overcoming the Pointer-Chasing Problem

Key:

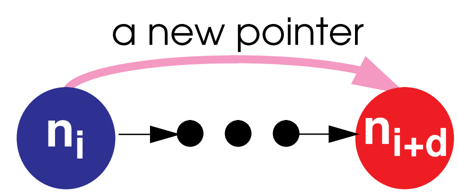
☞ n_i needs to know $\&n_{i+d}$ without referencing the d-1 intermediate nodes

Our proposals:

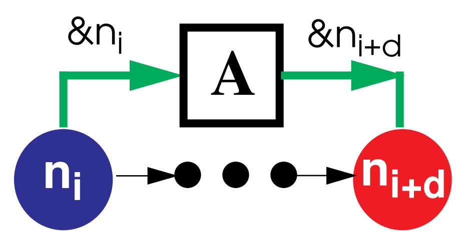
- use *existing* pointer(s) in n_i to approximate $\&n_{i+d}$
  - Greedy Prefetching



- 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.)
  - Data-Linearization Prefetching

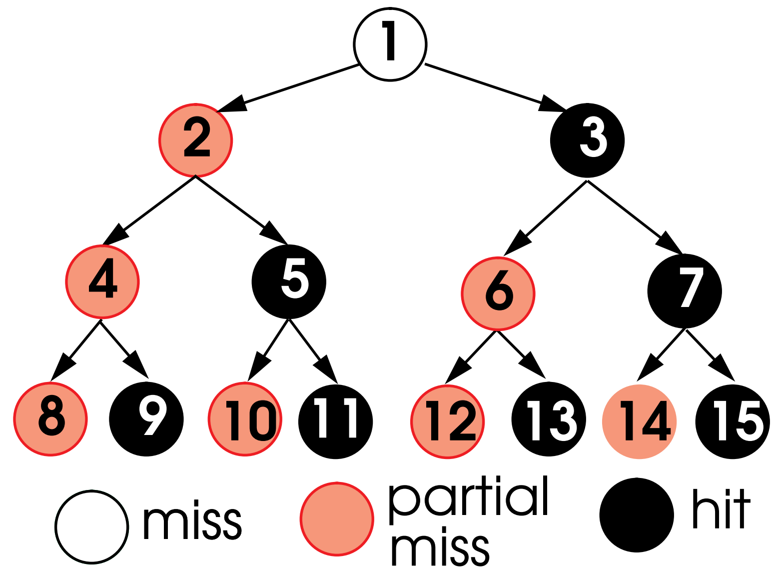


A=Address generating function

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);
   }
}
```

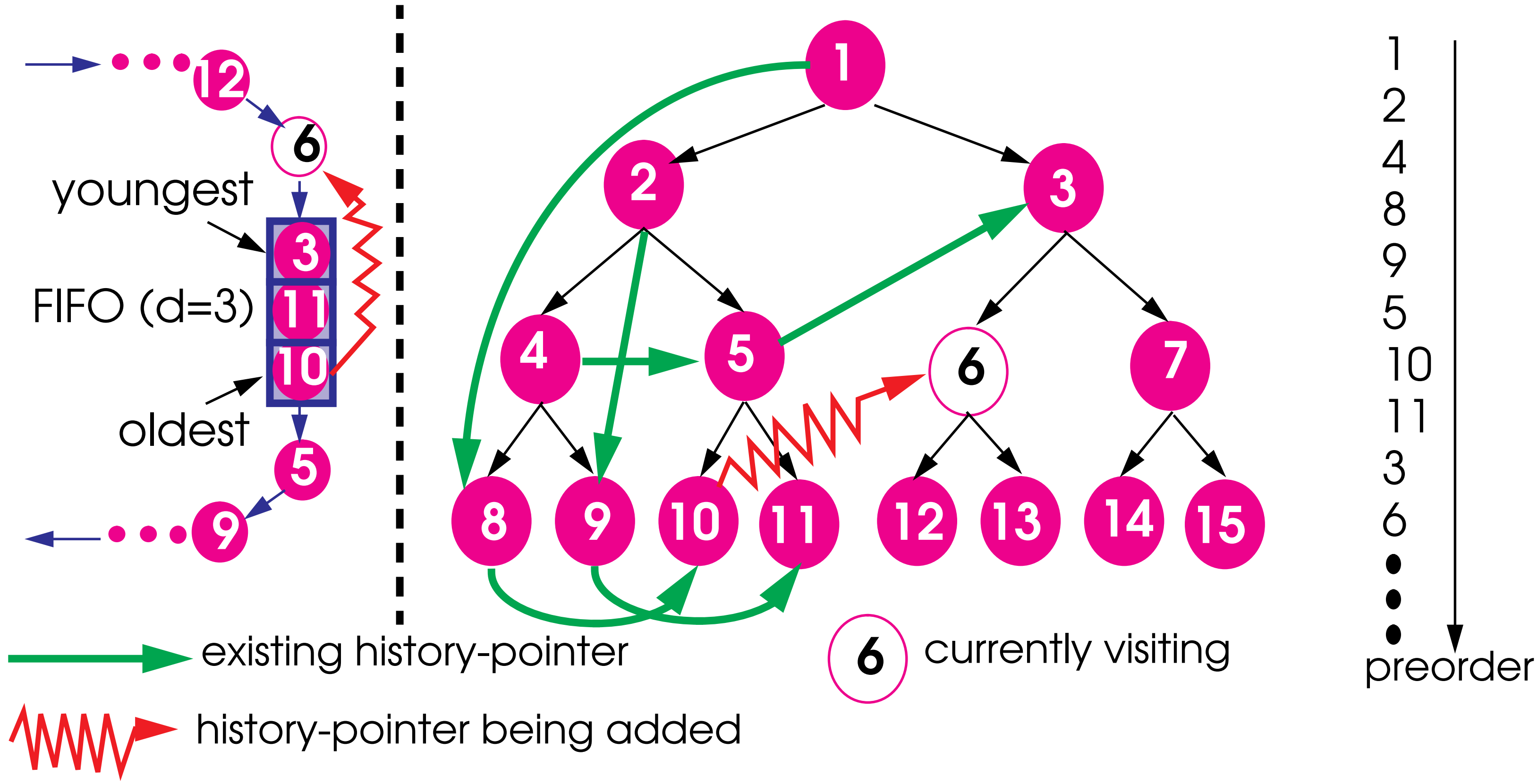


- **Reasonably effective in practice**

- **However, little control over the prefetching distance**

History-Pointer Prefetching

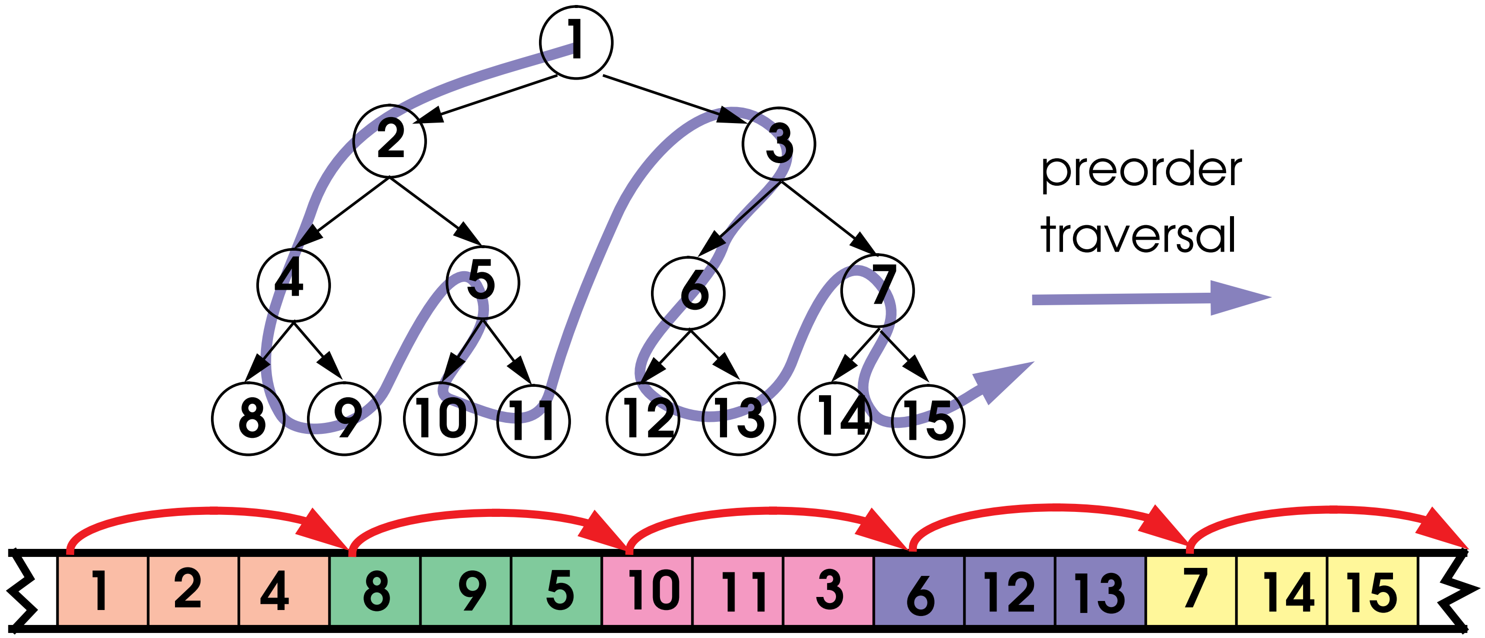
- **Add new pointer(s) to each node**
  - history-pointers are obtained from some recent traversal



- **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



prefetching distance= 3 nodes

prefetch

Question:

Can we remap nodes dynamically without breaking the program?

☞ ***Yes, we can do this with memory forwarding!***

Summary of Prefetching Algorithms

	Greedy	History-Pointer	Data-Linearization
Control over Prefetching Distance	little	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 if no dynamic remapping;

- **Greedy prefetching is the most widely applicable algorithm:**
  - ☞ **fully automated in SUIF**
- **History-pointer and data-linearization prefetching:**
  - ☞ **profiling-assisted versions are being implemented**

Experimental Framework

Benchmarks

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

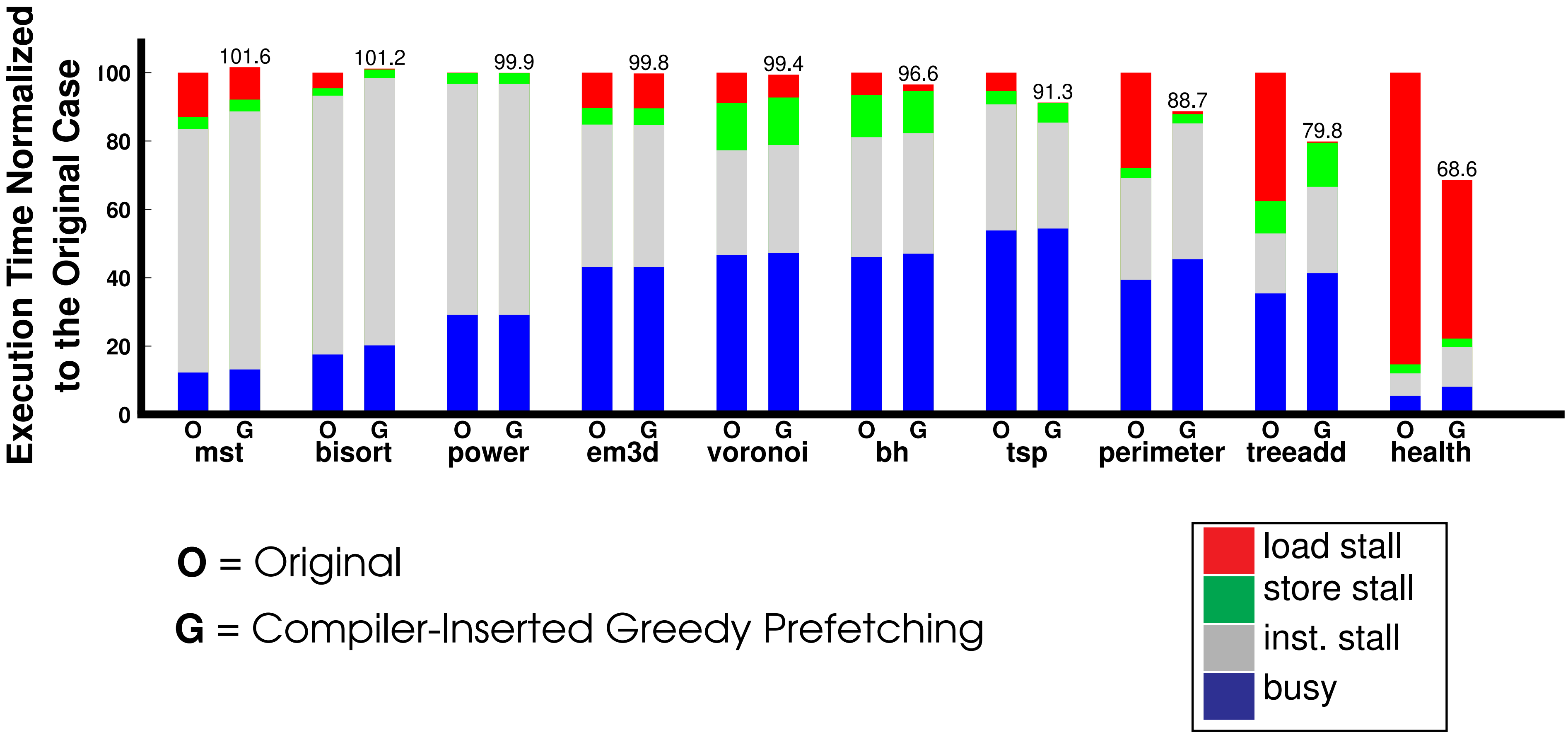
Simulation Model

- Detailed, cycle-by-cycle simulations of a MIPS R10000-like processor
  - 16 KB I-cache, 16 KB D-cache, 512 KB unified Secondary cache
  - miss penalties: 12 / 75 cycles

Compiler

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

Performance of Compiler-Inserted Greedy Prefetching



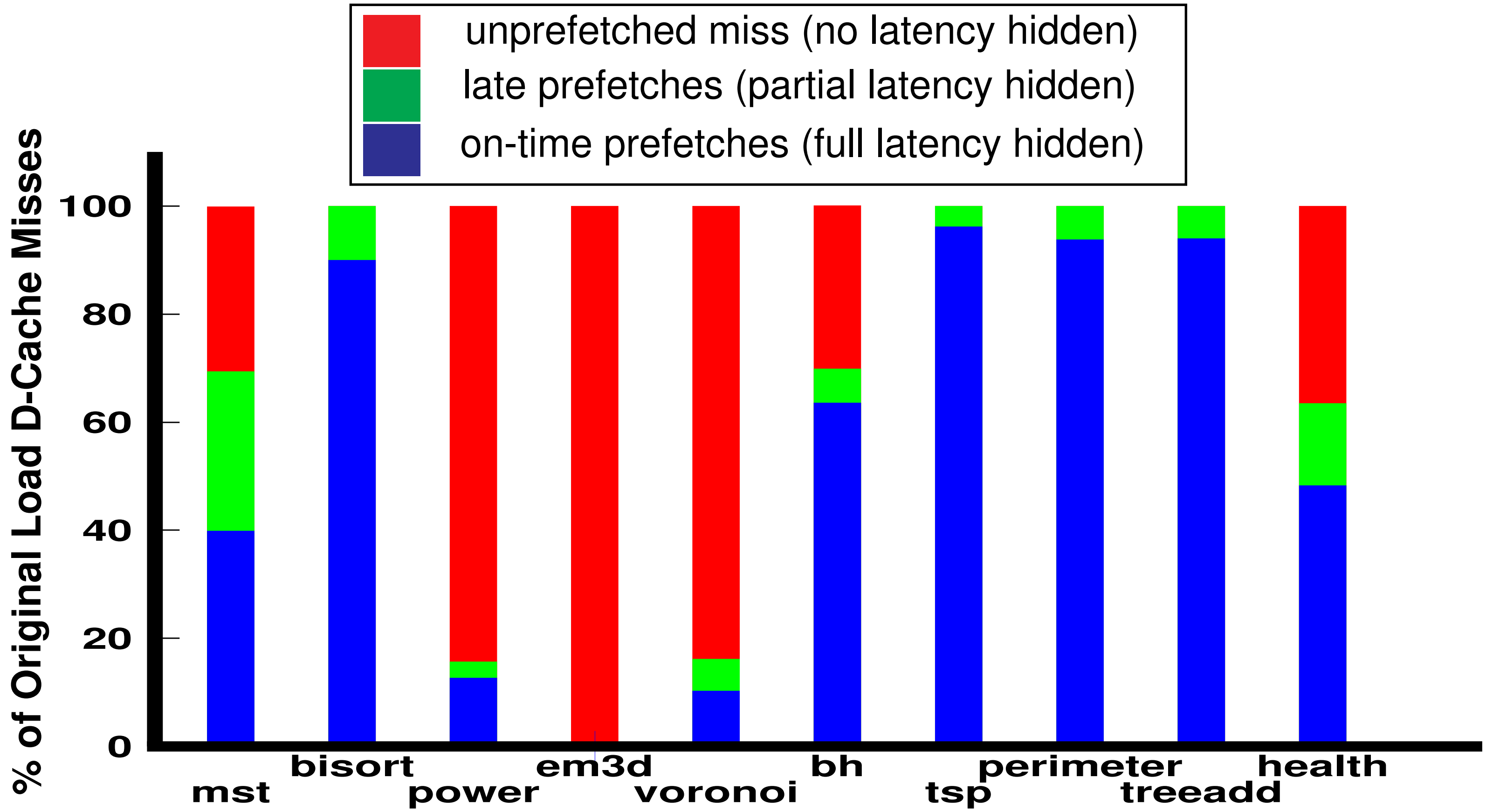
O = Original

G = Compiler-Inserted Greedy Prefetching

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

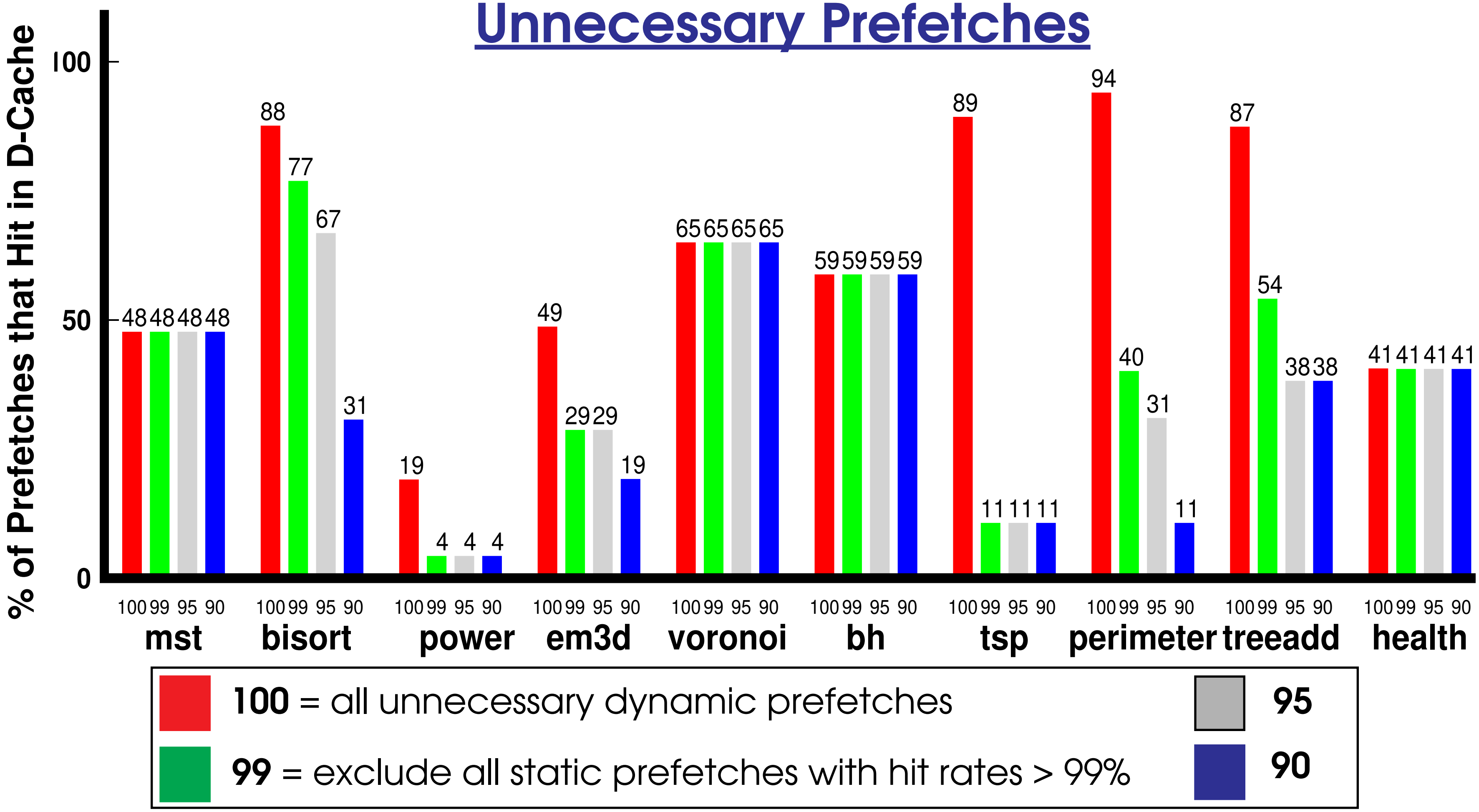
☞ **half achieve speedups of 4% to 45%**

Effectiveness of Greedy Prefetching



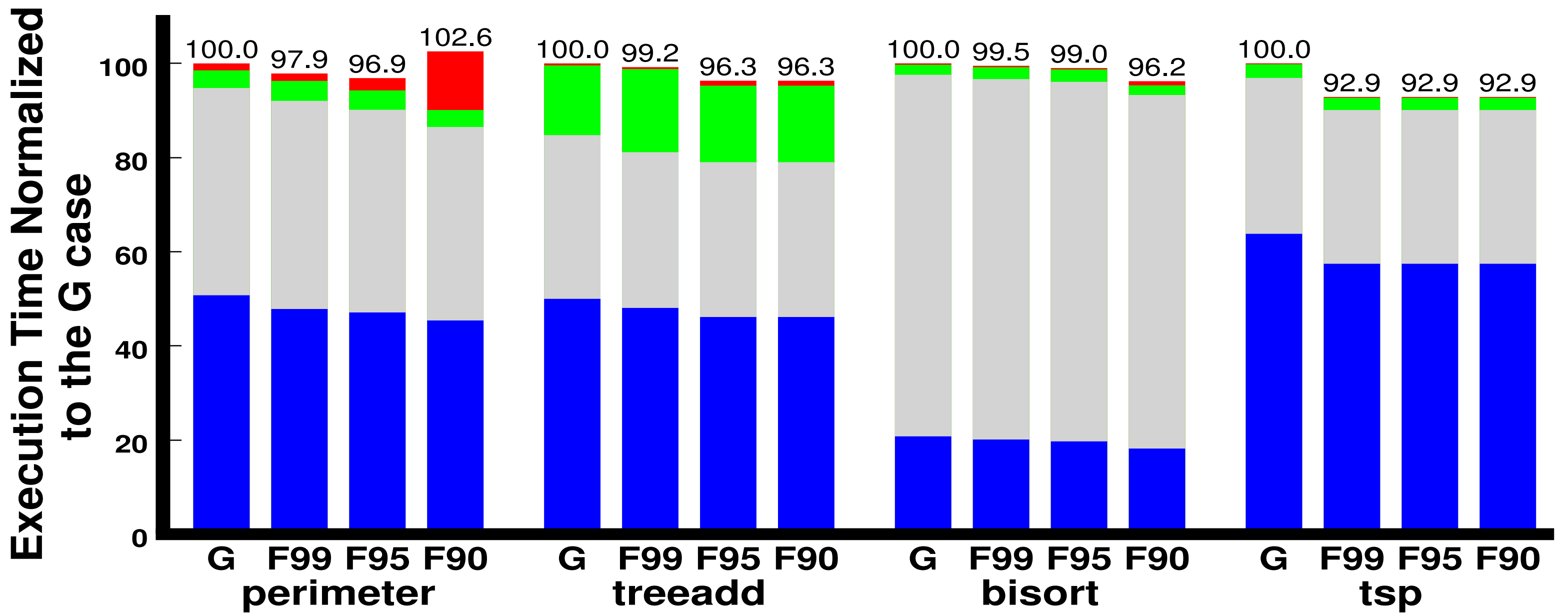
- 7 out of 10 have > 60% of original load misses prefetched
    - em3d, power, voronoi have many array or scalar load misses
- most prefetches are on time => fairly effective prefetch scheduling

Unnecessary Prefetches



- **% dynamic pfs that are unnecessary because the data is in the D-cache**
- **Bisort, tsp, perimeter, and treeadd have >80% unnecessary prefetches**
- **Could reduce overhead by eliminating static pfs that have high hit rates**

Reducing Overhead Through Memory Feedback



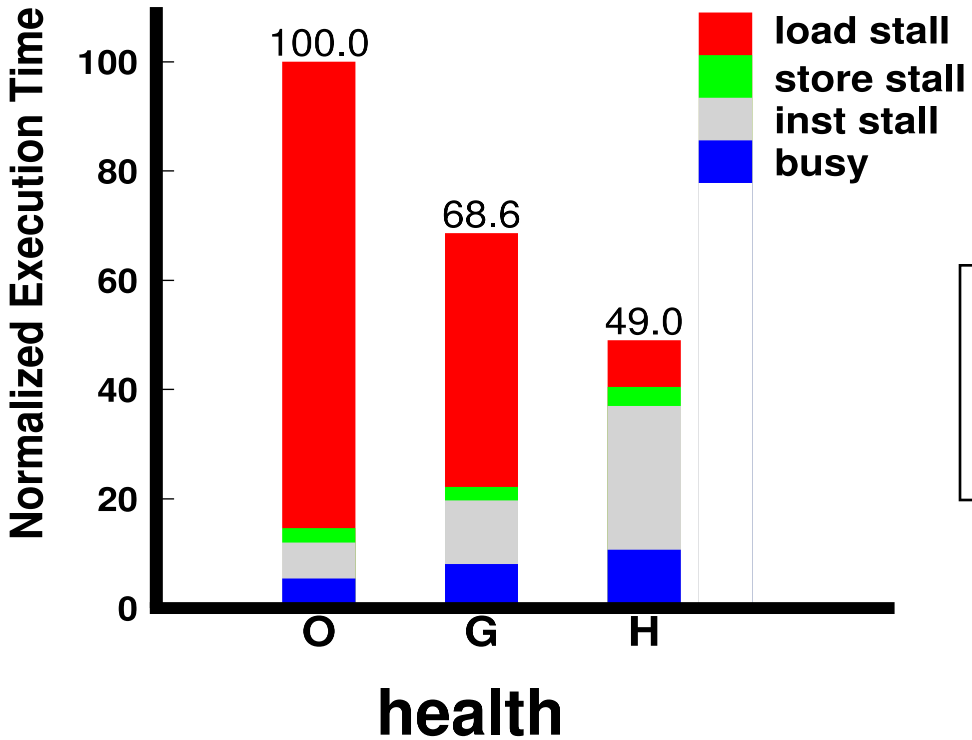
G = greedy prefetching

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

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

☞ **memory feedback can potentially improve performance**

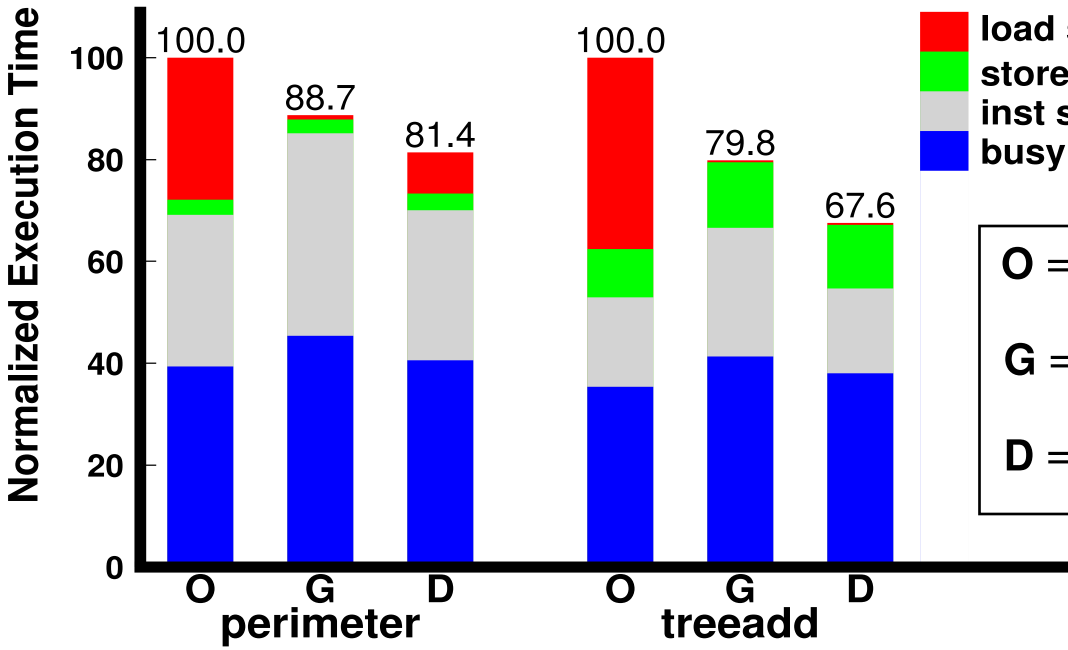
Performance of History-Pointer Prefetching



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

- **Applicable because a list structure does not change over time**
- **40% speedup over “greedy” (twofold speedup over no prefetching)**
- **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**

Overview

✓ Coping with Memory Latency

☞ Improving Data Cache Performance

  ✓ Compiler-Based Prefetching for Recursive Data Structures

  ☞ Memory Forwarding

- Improving Instruction Cache Performance
  - Cooperative Instruction Prefetching
- Conclusions

Data Locality Optimizations for Non-Numeric Codes

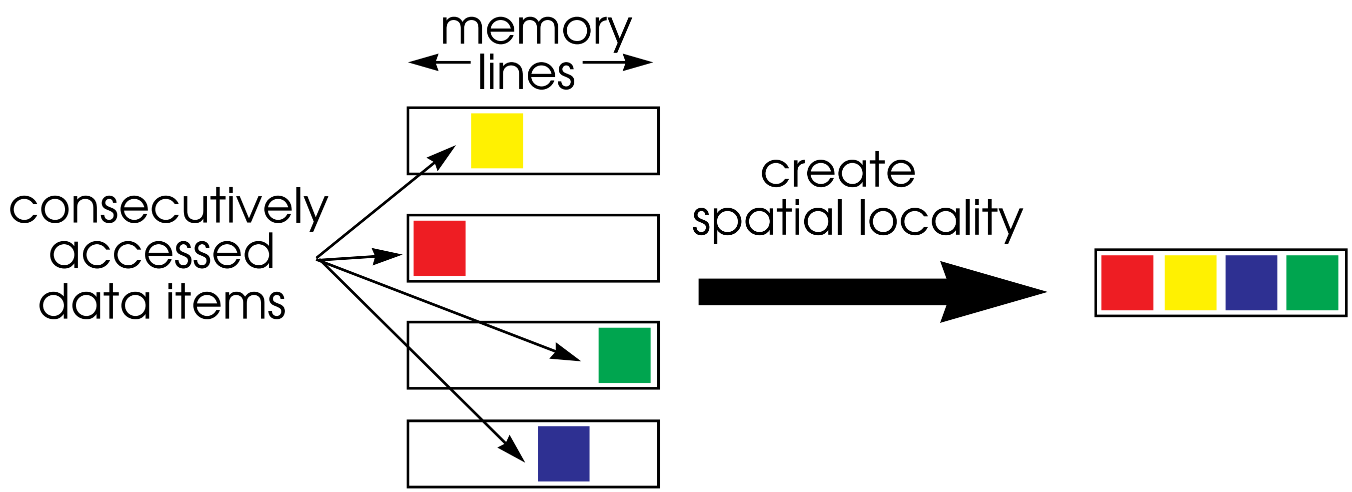
Two possibilities:

1. **Reorder computation**
   - given a fixed **data layout**

2. **Reorganize data layout**
   - given a fixed order of **computation**

☞ **our focus: use data layout optimizations to improve cache performance**

Layout Optimizations: Creating Spatial Locality

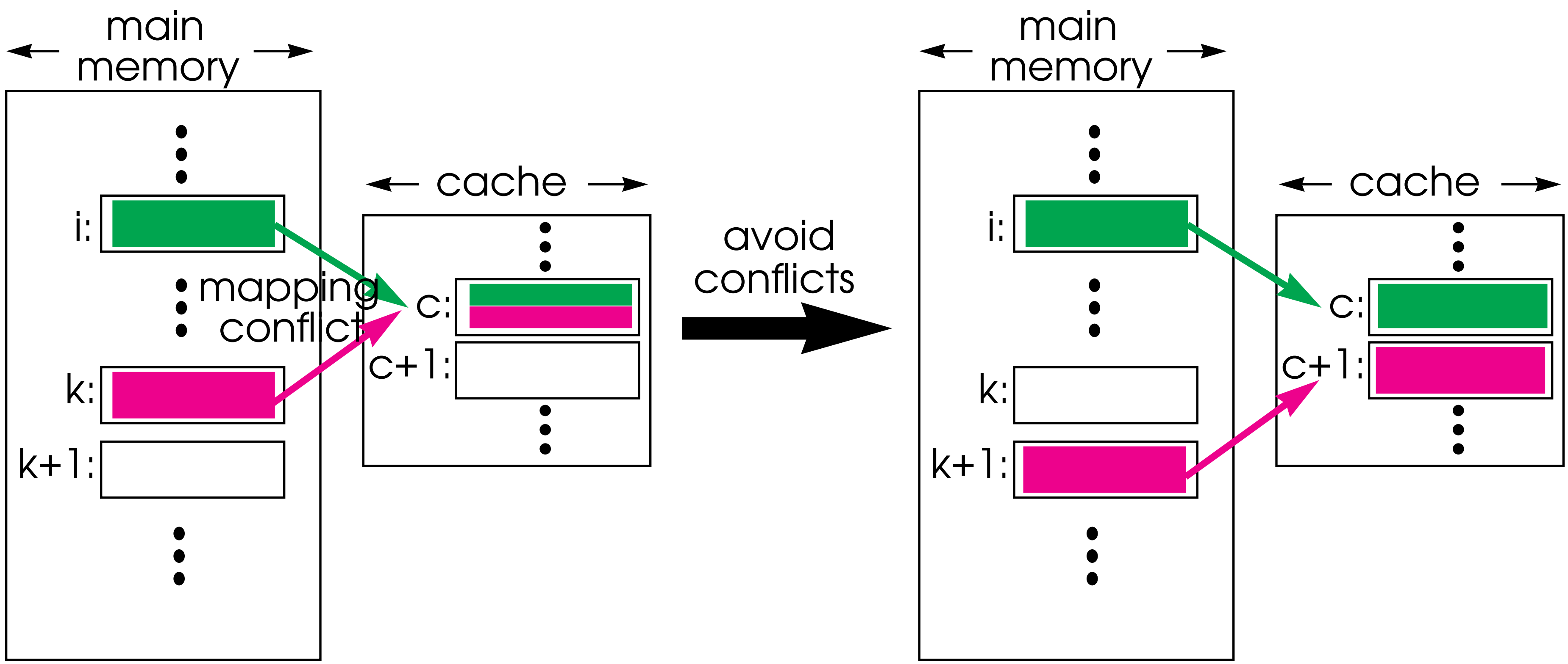


Importance of Spatial Locality:

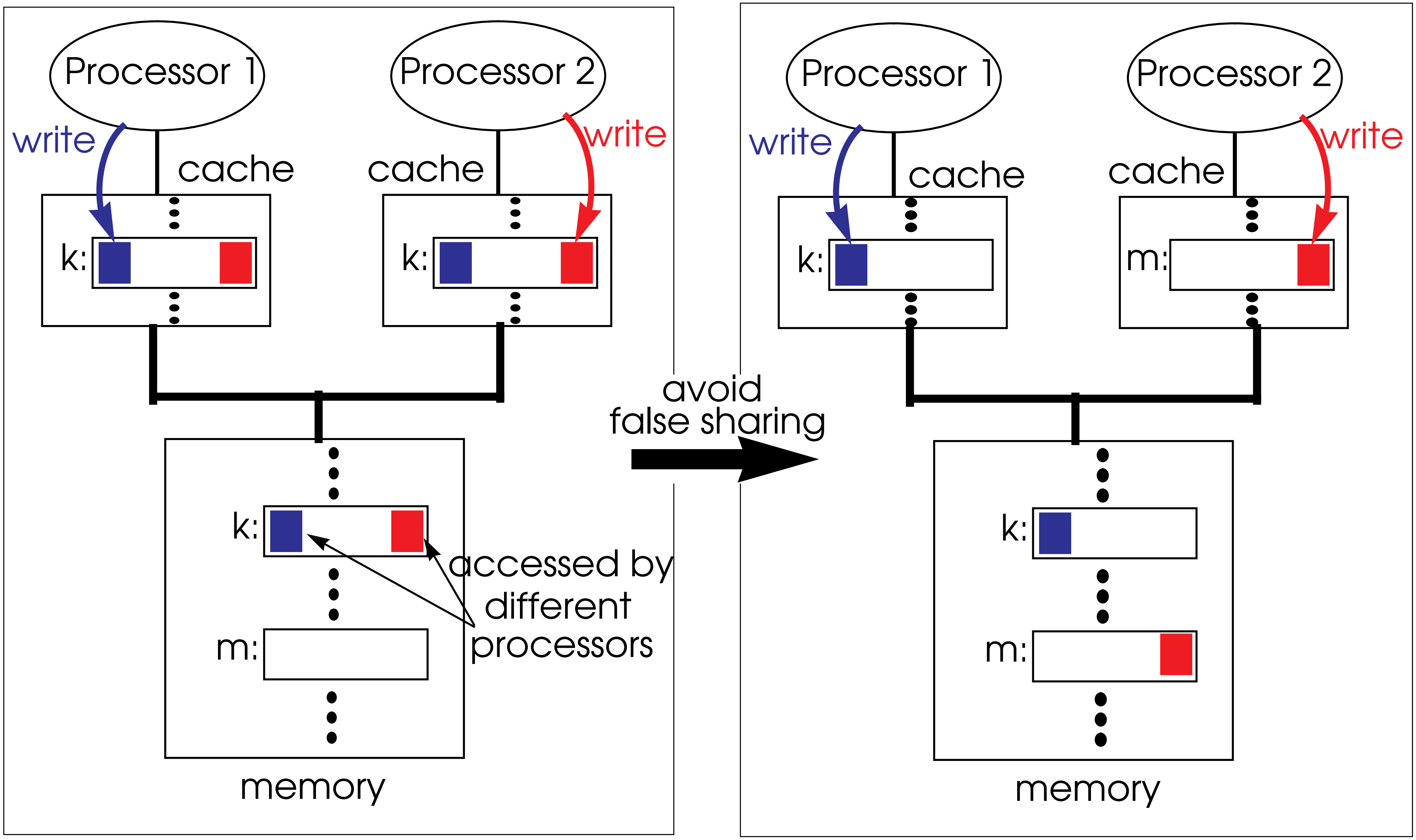
- reducing the number of cache misses
- reducing memory bandwidth consumption
- facilitating prefetch address generation

☞ **useful at all levels of the memory hierarchy**

Layout Optimizations: Avoiding Cache Conflicts



Layout Optimizations: Avoiding False Sharing



Static vs. Dynamic Layout Optimizations

	Data Layout Optimizations	
	Static	Dynamic
When Layout is Optimized	when data objects are created	can occur at any time during the execution (one or more times)
Advantage	relatively easy to implement	adapt to dynamic program behaviors
Key Challenge	find the "best" layout for the entire execution for all inputs	**guarantee the safety of data relocation**

Dynamic Layout Optimization Techniques

List Linearization (Clark '76):

- relocate nodes of linked lists to contiguous memory locations

Data Copying (Lam *et al* '91):

- copy a tile in a multi-dimensional array to contiguous locations

Tree Clustering (Chilimbi *et al* '99):

- pack subtrees into a cache line

Data Coloring (Chilimbi *et al* '99):

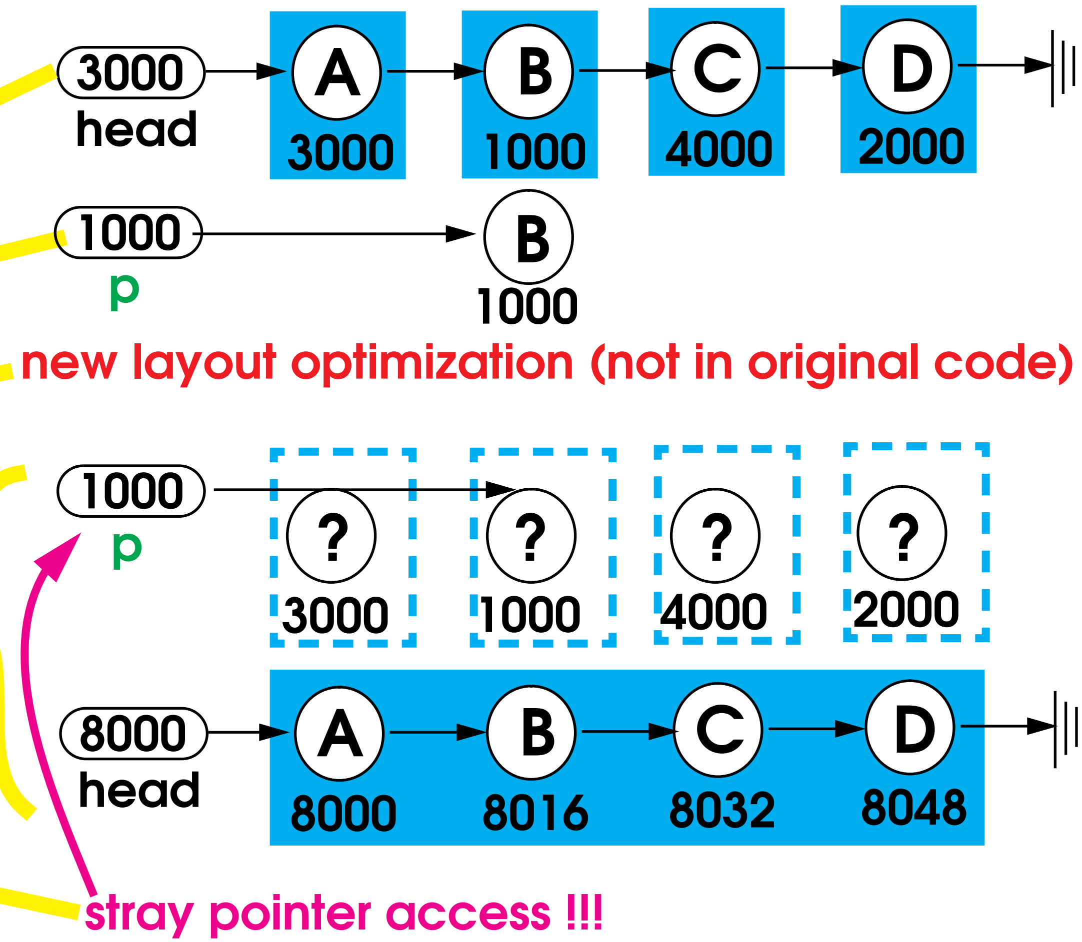
- map frequently-accessed elements to separate cache regions

Data-Linearization Prefetching (Luk and Mowry '96):

- map nodes close in the traversal to contiguous memory locations

List Linearization: Example

```
list * head, *p;
      :
traverse(head);
      :
p = head->next;
      :
list_linearize(&head);
      :
traverse(head);
      :
 if (debug()){
    print(p->data);
 }
```



☞ need techniques to guarantee the safety of data relocation

Current Approaches to Relocation Safety

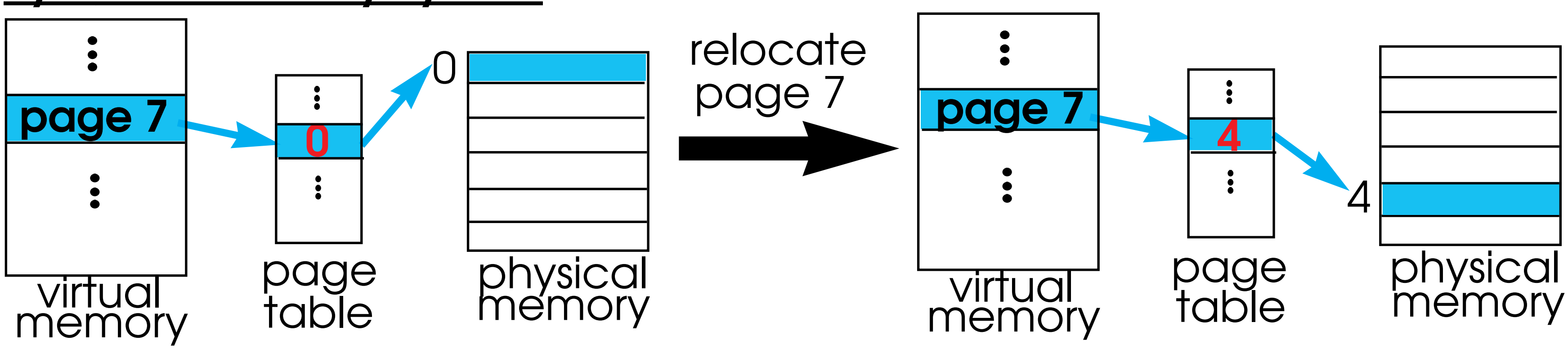
By Programmer:

- error prone for large programs

By Compiler:

- perfect aliasing information is impractical
- run-time behaviors are not known
  - ☞ must be conservative

By Virtual Memory System:



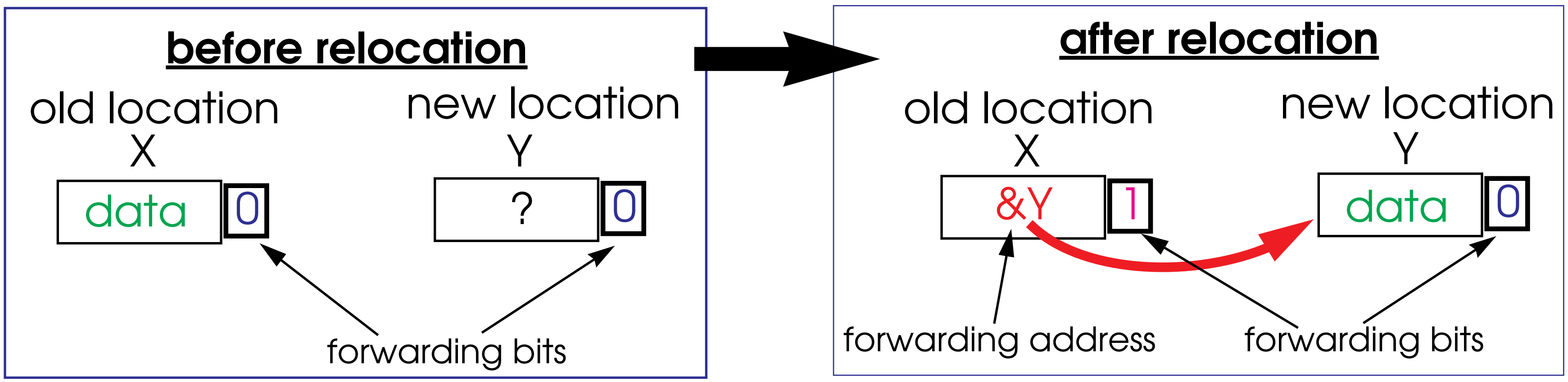
- provides safe data relocation at a **page granularity**
- however, to improve cache perf., we must relocate at a **word granularity**
  - setting the page size to be one word is *not* viable

☞ **our solution:** ***memory forwarding***

Memory Forwarding

Basic Idea:

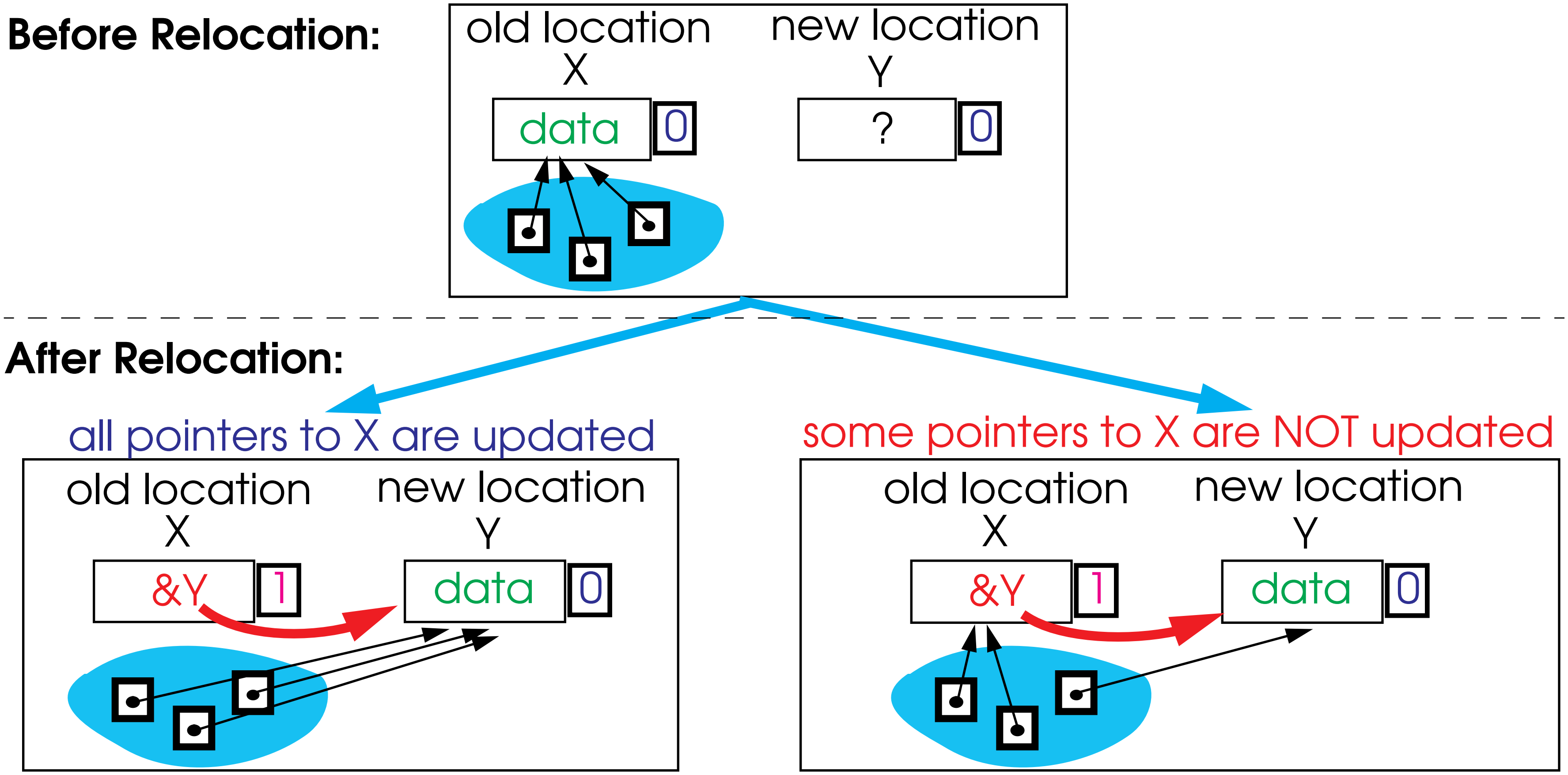
☞ **add a layer of indirection *within* the memory system**



- to relocate an object:
  - copy it into the new location
  - store the new address into the old location
  - mark the old location as containing a forwarding address
- software attempts to update all affected pointers with the new address
- in case the old location is accessed by a stray pointer:
  - hardware automatically forwards the access to the new location

☞ **memory forwarding functions as a safety net**

Speculating that Pointers Have Been Updated

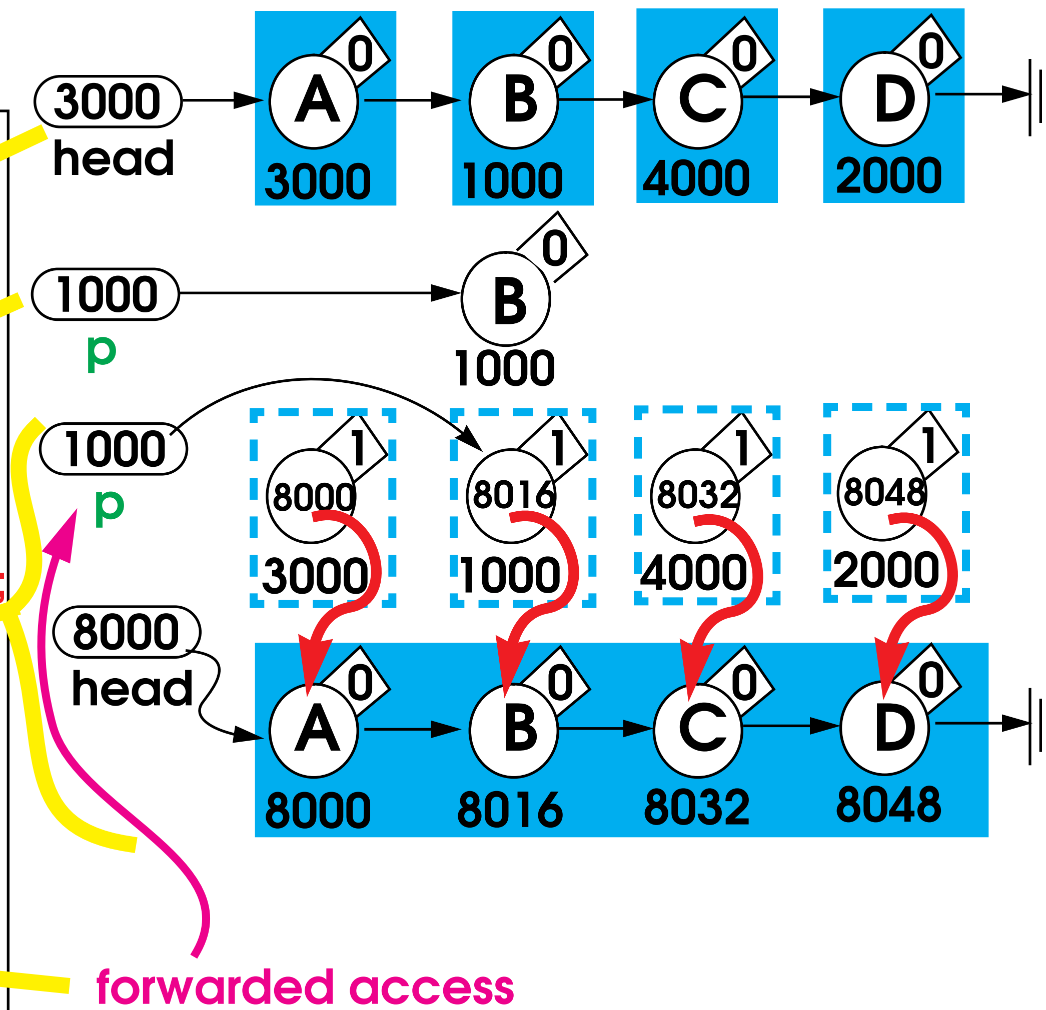


- **Software speculates that all pointers to the old location are updated**
- **If wrong, there is a recovery cost but the execution is still correct**

☞ **performance is improved only if the speculation is correct often enough**

List Linearization (With Memory Forwarding)

```
list * head, *p;
        ⋮
traverse(head);
        ⋮
p = head->next;
        ⋮
list_linearize_forward(&head);
        ⋮
traverse(head);
        ⋮
 if (debug()){
   print(p->data);
 }
```

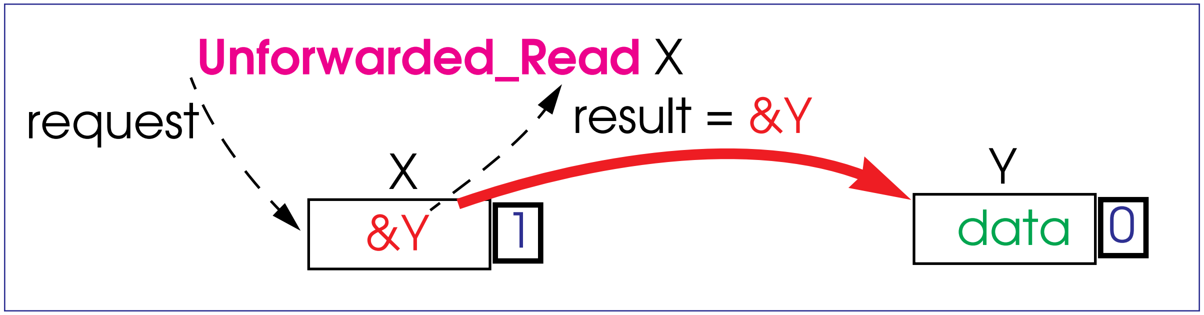


Memory Forwarding: Implementation Issues

- Extensions to the Instruction Set Architecture (ISA)
- Hardware Support
- Software Support

Extensions to the ISA

- R = Read_FBit(word* A)
  - reads the forwarding bit of the word at address **A** into register **R**
- R = Unforwarded_Read(word* A)
  - reads the content of word **A** into **R**, with forwarding disabled



- *A = Unforwarded_Write(word R, bit B)
  - performs these two actions together atomically:
    - writes the value of **R** into word **A**, with forwarding disabled
    - sets the forwarding bit of word **A** to **B**

Hardware Support

Key Insight:

- actual forwarding rarely occurs
  - because software has updated most affected pointers properly

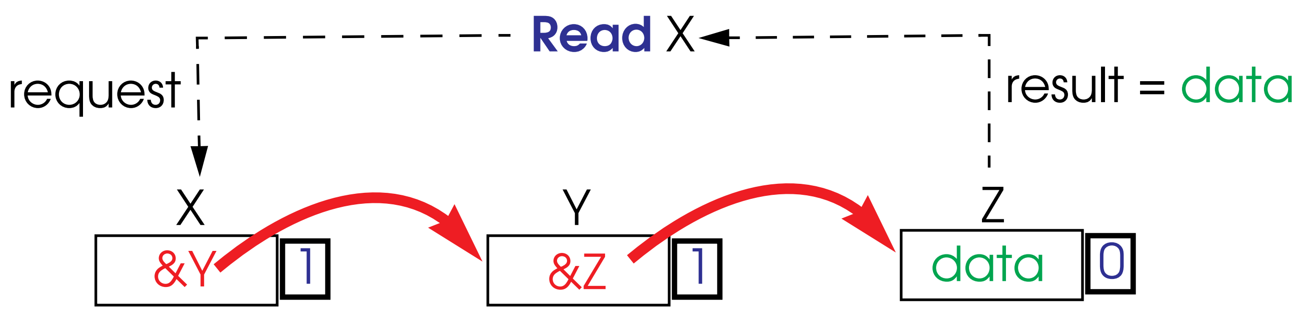
☞ can implement forwarding as an **exception** in modern processors

New Features:

- tagged memory (1 bit / word)
- dereferencing forwarding addresses
- handling forwarding cycles
- providing user-level traps upon forwarding
- data dependence speculation

Dereferencing Forwarding Addresses

Example of Dereferencing Forwarding Addresses:

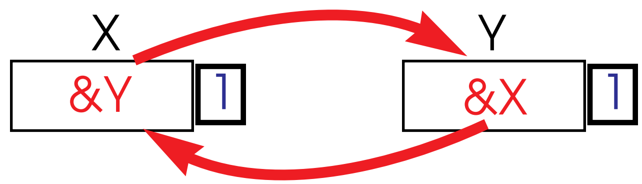


Two Implementation Options:

- purely in hardware
- use a software-based exception handler:
  - exception triggered by accessing a word with a set forwarding bit
  - handler uses our ISA extensions to chase forwarding chain

Handling Forwarding Cycles

Example of a forwarding cycle:



- **Forwarding cycles are unexpected but can be caused by software bugs**
- **Accurate detection of forwarding cycles is too expensive**

One Inexpensive Approach:

- use a counter to keep track of the # of forwarding hops performed
- if counter > limit, raise an exception to check if there is a cycle:
  - not a cycle => reset the counter and resume execution
  - is a cycle => abort the execution

Providing User-Level Traps Upon Forwarding

- **May be helpful to provide a lightweight user-level trapping mechanism:**
  - similar to informing memory operation traps (Horowitz *et al.* '96)
    - triggered by cache misses
  - our user-level traps are invoked upon accessing forwarded locations

Uses:

- as a profiling tool to gather forwarding-related statistics
  - e.g., record which instructions experienced forwarding (to eliminate that forwarding in future runs)

  ☞ improve performance of future execution

- as a run-time optimization tool to eliminate forwarding on-the-fly
  - e.g.,update stray pointers to point directly to the new address

  ☞ improve performance of the current execution

Software Support

- **Initialization of Forwarding Bits**
- **Deallocation of Forwarded Data**
- **Memory Alignment of Relocatable Data Objects**
- **Preserving Outcomes of Pointer Comparisons**
  - outcomes of pointer comparisons can be changed by relocation
    - pointer analyses help detect which comparisons could be affected
  - with forwarding, compiler can insert explicit codes to make sure that:
    - comparisons are performed with respect to the final addresses (these final address can be computed using our ISA extensions)

Example:

```
node *p, *q;
...
if (p == q)
...
```

compiler inserts codes →

```
node *p, *q;
...
if (final_addr(p) == final_addr(q))
...
```

Experimental Framework

Benchmarks

- 7 non-numeric applications
- various aggressive data layout optimizations applied
  - e.g., list linearization, tree clustering, packing hash table elements
  - compiler can't prove their legality
  - fortunately forwarding guarantees that they are safe to apply

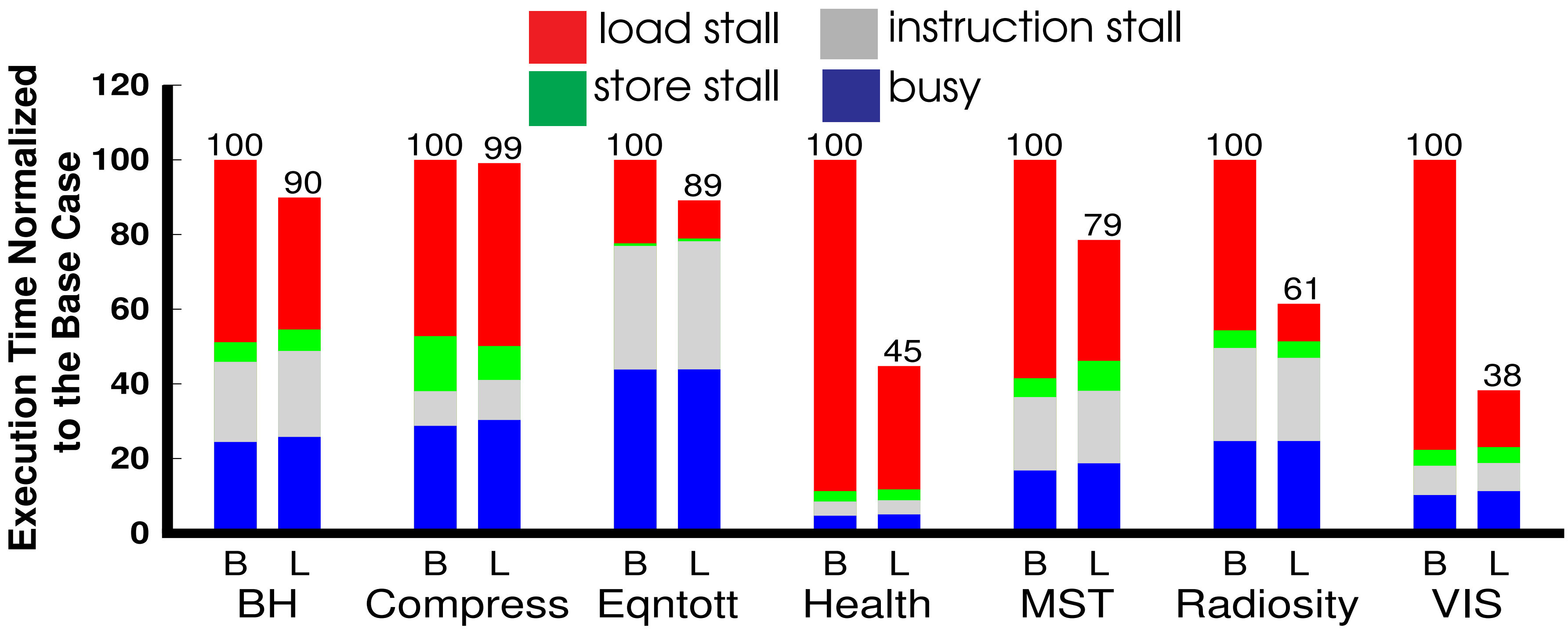
Simulation Model

- detailed, cycle-by-cycle simulations of a MIPS R10000-like processor
  - 16 KB I-cache, 16 KB D-cache, 512 KB unified Secondary cache
  - Line sizes: 32B - 512B
- includes all hardware and software support for memory forwarding

Compiler

- MIPS CC with -O2 under IRIX 5.3

Performance of Forwarding-Enabled Layout Optimizations



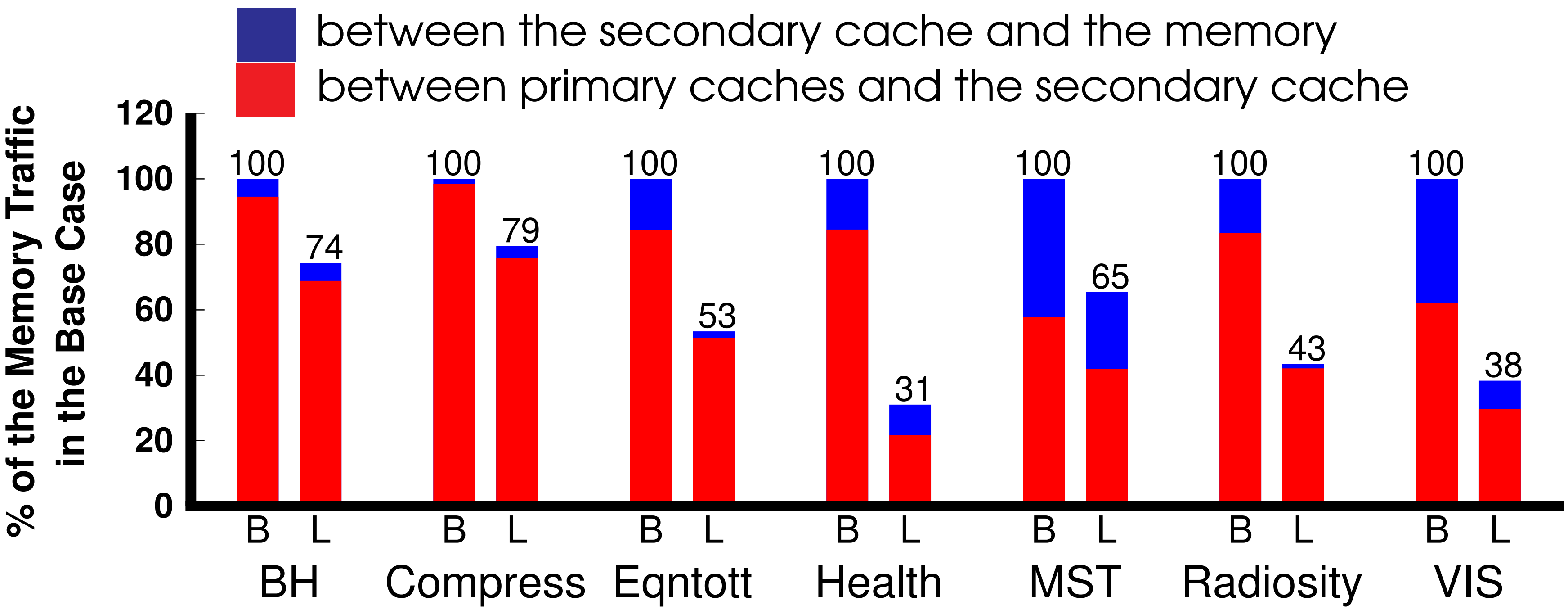
B = Base Case **L** = Layout Optimized

line size = **128B** (except in BH, where line size = **512B**)

- **Layout optimizations improve spatial locality quite dramatically**

☞ **achieve large speedups (> twofold in Health and VIS)**

Impact on Memory Bandwidth Consumption



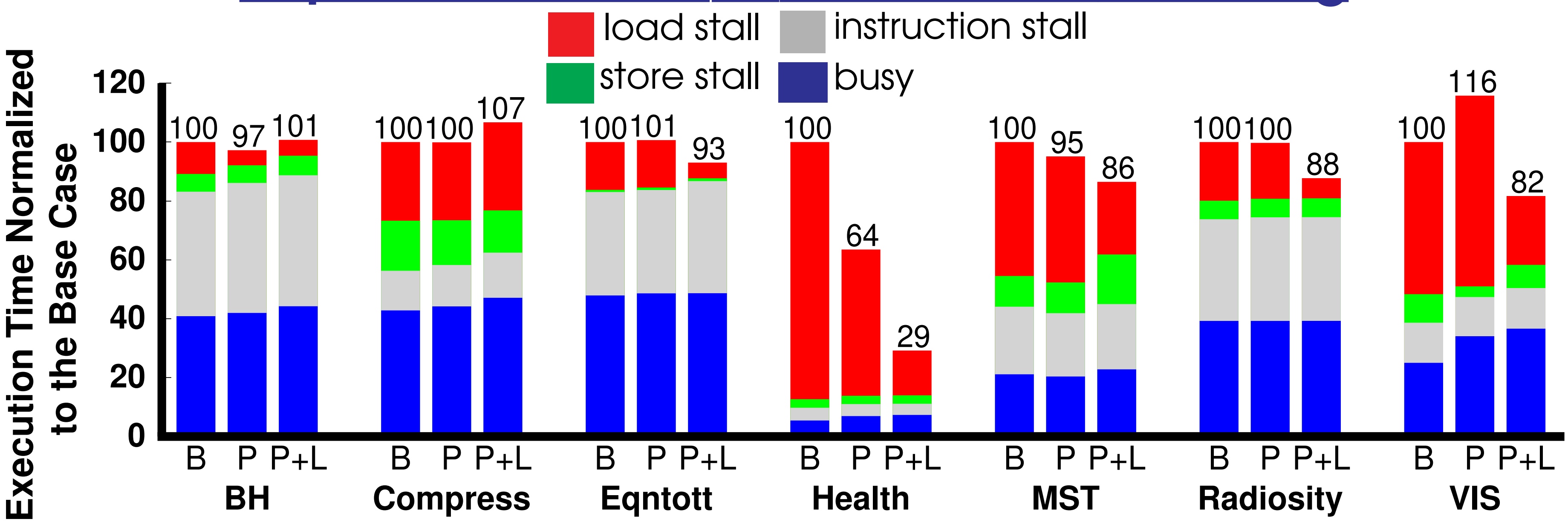
B = Base Case **L** = Layout Optimized

line size = **128B** (except in BH, where line size = **512B**)

- **Layout-optimized code consumes 21%-69% less bandwidth**

☞ **deliver higher performance *and* reduced bandwidth consumption**

Impact on the Performance of Prefetching



B = Base Case
P+L = Prefetching Plus Layout Optimizations
P = Prefetching Without Layout Optimizations
line size = **32B**

- **Layout optimizations improve performance of prefetching in 5 programs**
- **Health, MST, Radiosity, VIS:**
  - operate on linked lists that could be inserted/deleted frequently
  - **P+L** is essentially data-linearization prefetching
    - ☞ layout optimizations alleviate the pointer-chasing problem

☞ **forwarding broadens the scope of data-linearization prefetching**

Overview

✓ Coping with Memory Latency

✓ Improving Data Cache Performance

☞ Improving Instruction Cache Performance

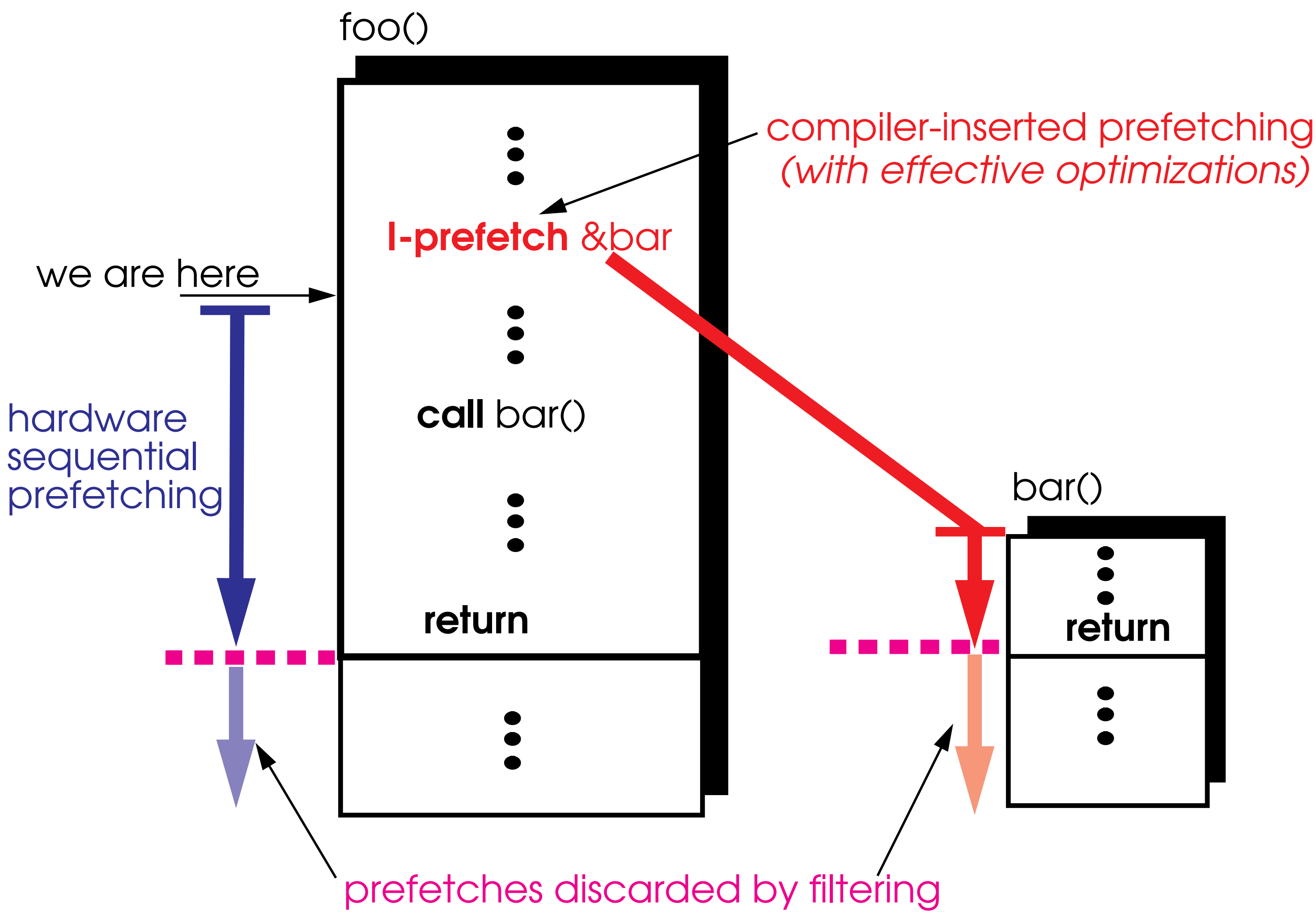
- ☞ Cooperative Instruction Prefetching

• Conclusions

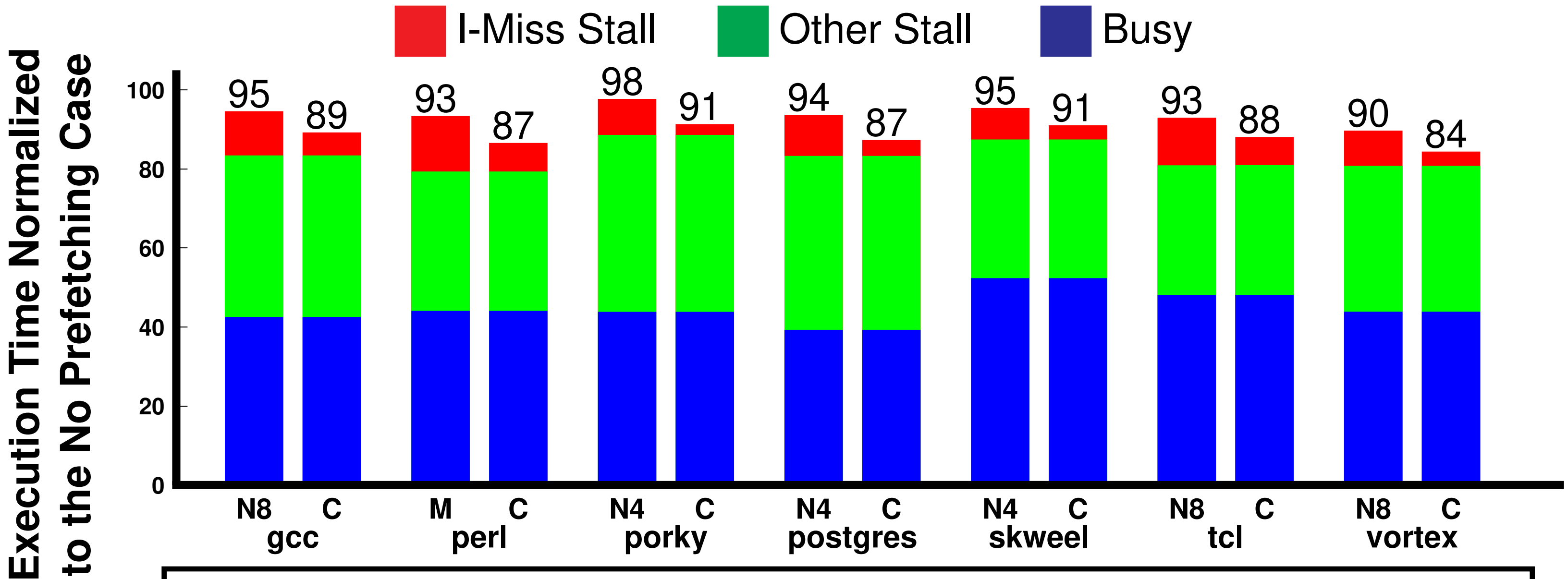
Cooperative Instruction Prefetching

Goal:

prefetch far enough in advance with few useless prefetches



Cooperative Prefetching vs. Best Previous Schemes



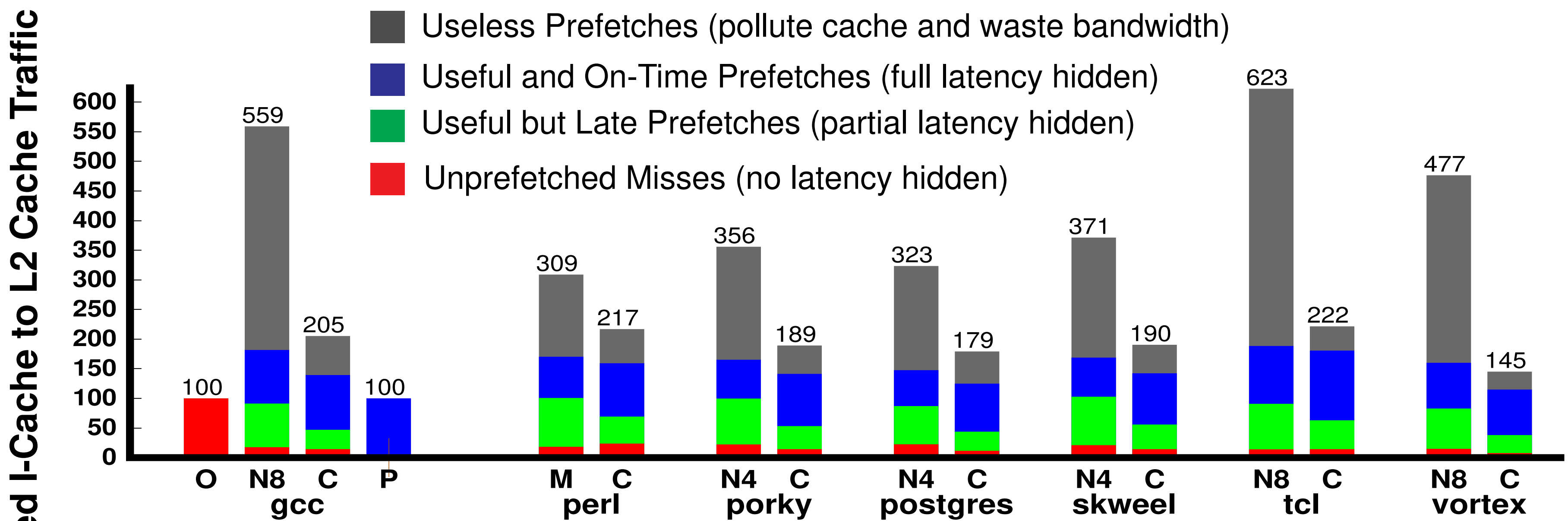
Best previous schemes:

N*x* = Next-*x*-line prefetching
M = Markov prefetching

C = Cooperative prefetching

- Cooperative prefetching reduces **50% or more** of remaining I-miss stall
  - ☞ **achieves 13.3% speedup over no prefetching**
    - double the speedup (6.5%) offered by best previous schemes
    - much closer to the maximum 20% speedup
  - ☞ **achieves 24.4% speedup when latency is increased to 24 cycles**

Prefetching Effectiveness



O = Original **P** = Perfect prefetching

C = Cooperative prefetching

Best previous schemes:

N*x* = Next-*x*-line prefetching

M = Markov prefetching

Advantages of Cooperative Prefetching over Best Previous Schemes:

- Converting ~50% of useful but late prefetches into on-time prefetches
- Reducing amount of useless prefetches by a factor of 6

☞ **cooperative prefetching is far more effective than previous schemes**

Conclusions

Data Access Latency:

- **compiler-based prefetching for pointer-based codes**
  - 3 schemes to overcome the pointer-chasing problem:
    - greedy (fully automated), history-pointer, data-linearization
    - offer significant performance improvement
- **memory forwarding**
  - frees layout optimizations from concerns over breaking the program
    - enables impressive performance gains (and consumes less bw.)
  - can be implemented as an exception in modern processors

Instruction Access Latency:

- **cooperative instruction prefetching**
  - hardware prefetches aggressively and filters away useless prefetches
  - compiler prefetches non-sequential accesses far enough in advance
  - offers performance much closer to the ideal than previous schemes

☞ ***This dissertation provides viable solutions to the key problems that hinder the cache performance of non-numeric codes***