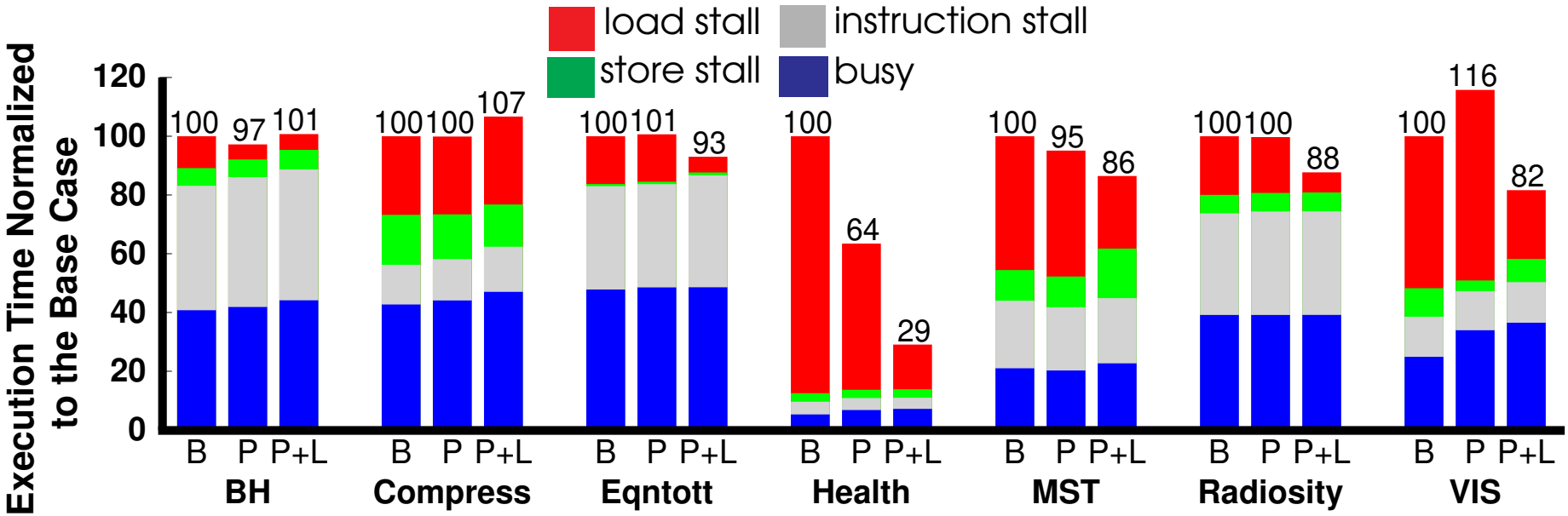
Conclusions

- **Dynamic Data Layout Optimizations:**
  - can increase the effectiveness of caches
  - but their correctness is difficult to guarantee in C/C++ programs

- **Memory Forwarding:**
  - frees layout optimizations from concerns over breaking the program
  - can be implemented as an exception in modern processors
  - enables significant performance gains and consumes less bandwidth
    - over twofold speedups in 2 applications
  - should be used carefully to avoid forwarding overhead
    - a user-level trap mechanism can identify and avoid forwarding

☞ ***should consider incorporating memory forwarding in the memory hierarchies of future computer systems***

Impact on the Performance of Prefetching

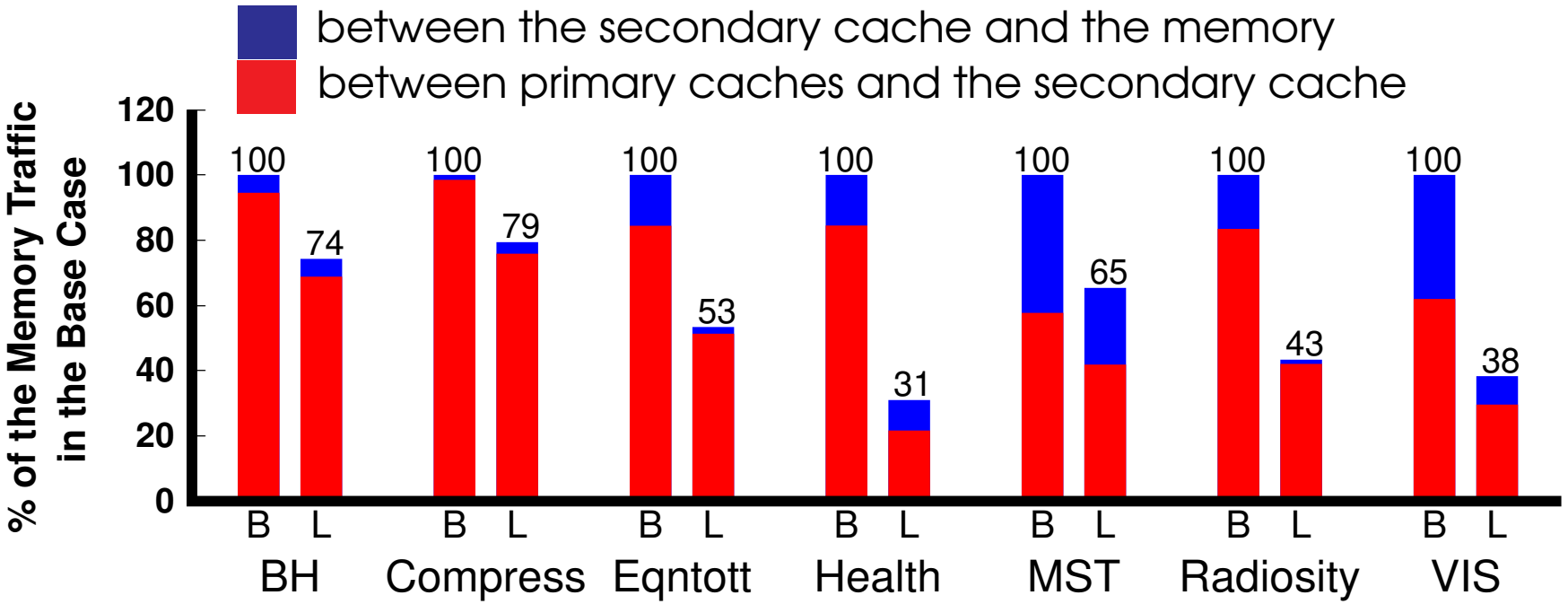


B = Base Case
P = Prefetching Without Layout Optimizations
P+L = Prefetching Plus Layout Optimizations
line size = **32B**

- **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 (Luk and Mowry '96)

☞ **layout optimizations facilitate prefetch address generation**

Impact on Memory Bandwidth Consumption



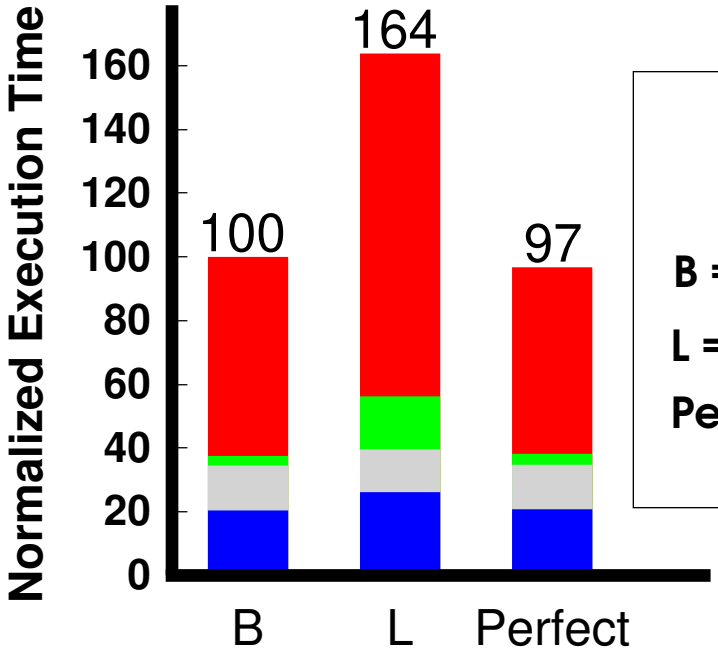
B = Base Case **L** = Layout Optimized

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 of Forwarding Overhead on an Application: SMV



load stall
instruction stall
store stall
busy

B = Base Case

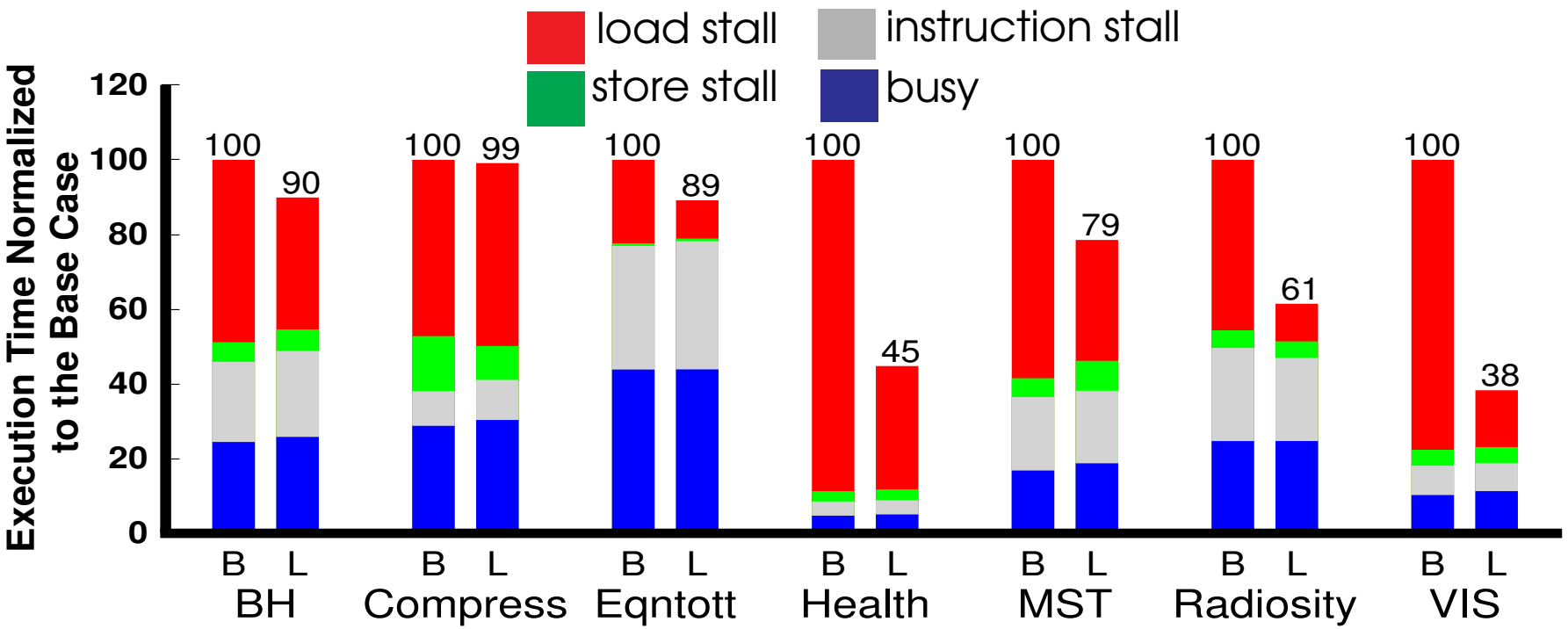
L = Layout Optimized with Realistic Forwarding

Perfect = Layout Optimized with Perfect Forwarding (i.e. without any forwarding overhead)

- **Nodes are connected both through a hash table and through trees**
- **Layout is only optimized for the hash table accesses**
- **Software can't update tree pointers to point to the relocated addresses**
  - ☞ forwarding occurs when relocated nodes are accessed via trees

☞ **a profiling tool based on user-level traps can help avoid such cases**

Performance of Forwarding-Enabled Layout Optimizations



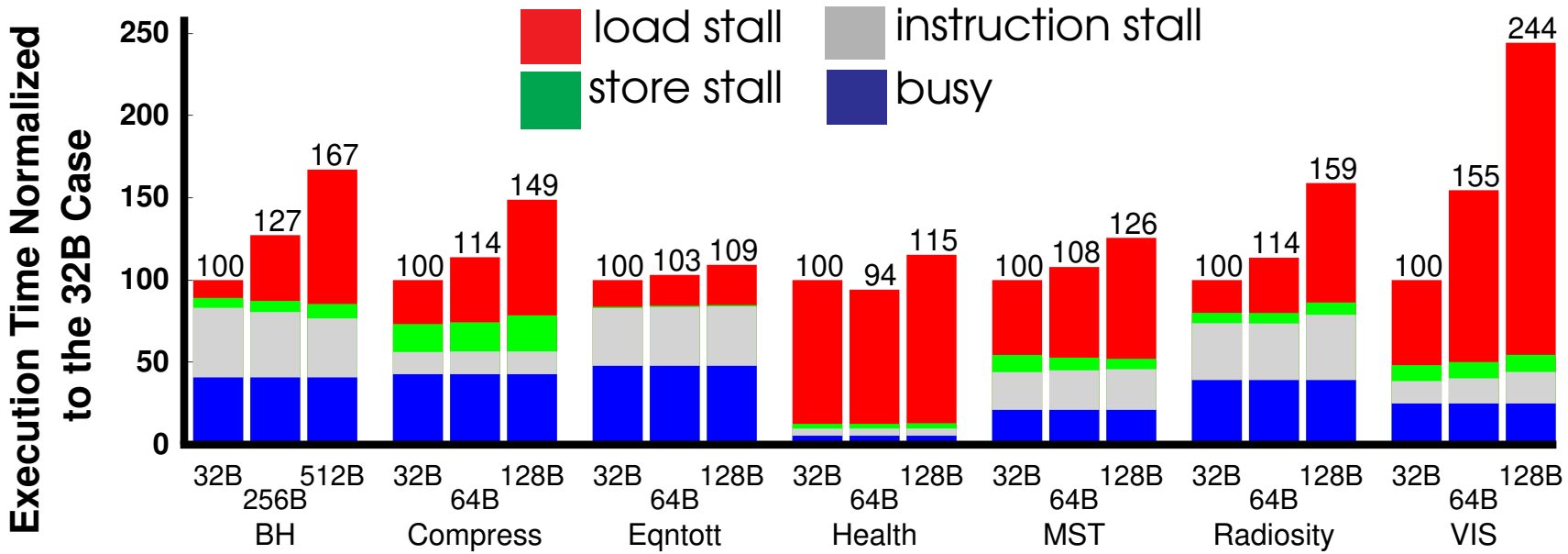
B = Base Case **L** = Layout Optimized

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)

Performance Impact of Cache Line Size



***x*B** = with *x*-byte cache lines

- **Performance degrades as we use longer line sizes**
  - ☞ lack of spatial locality in these irregular codes
- **However, longer line sizes do have other advantages**

Experimental Framework

Benchmarks

- 8 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

Overview

- ✓ Dynamic Data Layout Optimizations
- ✓ Memory Forwarding: Basic Concepts
- ✓ Implementation Issues
- ☞ Experimental Evaluation
- • Conclusions

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))
...
```

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)

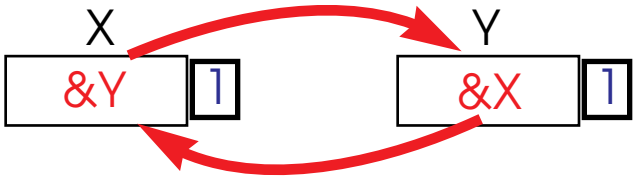
    ☞ improve performance of future execution

- 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

Handling Forwarding Cycles

Example of a forwarding cycle:



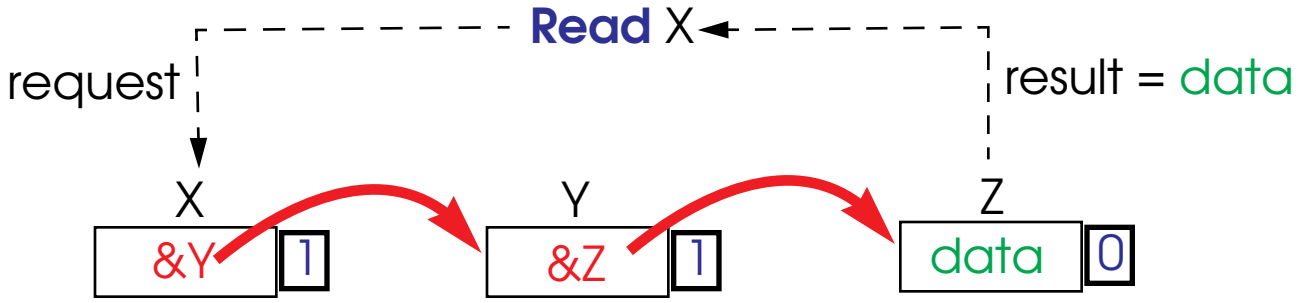
- **Forwarding cycles are unexpected but can be caused by software bugs**
- **Accurate detection of forwarding cycles is too expensive**

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

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

Hardware Support

Key Insight:

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

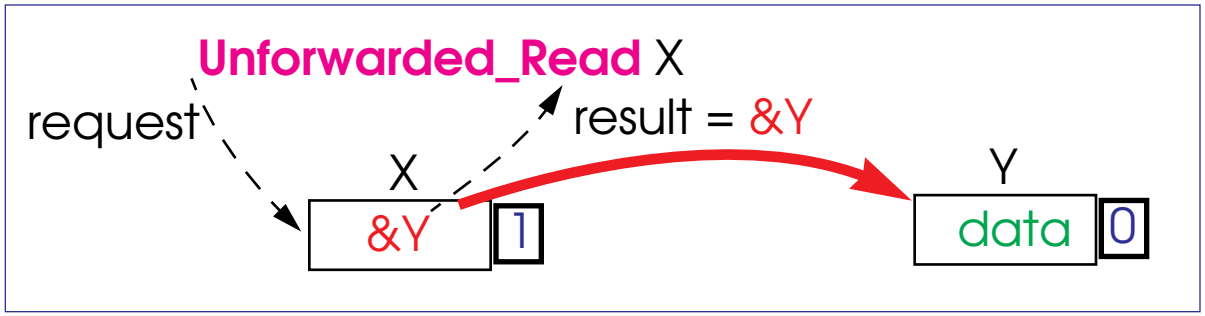
☞ can implement forwarding as an **exception** in modern processors

New Features:

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

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**

Overview

- ✓ Dynamic Data Layout Optimizations
- ✓ Memory Forwarding: Basic Concepts
- ☞ Implementation Issues
  - ☞ Extensions to the Instruction Set Architecture (ISA)
  - Hardware Support
  - Software Support
- Experimental Evaluation
- Conclusions

Memory Forwarding vs. Garbage Collection

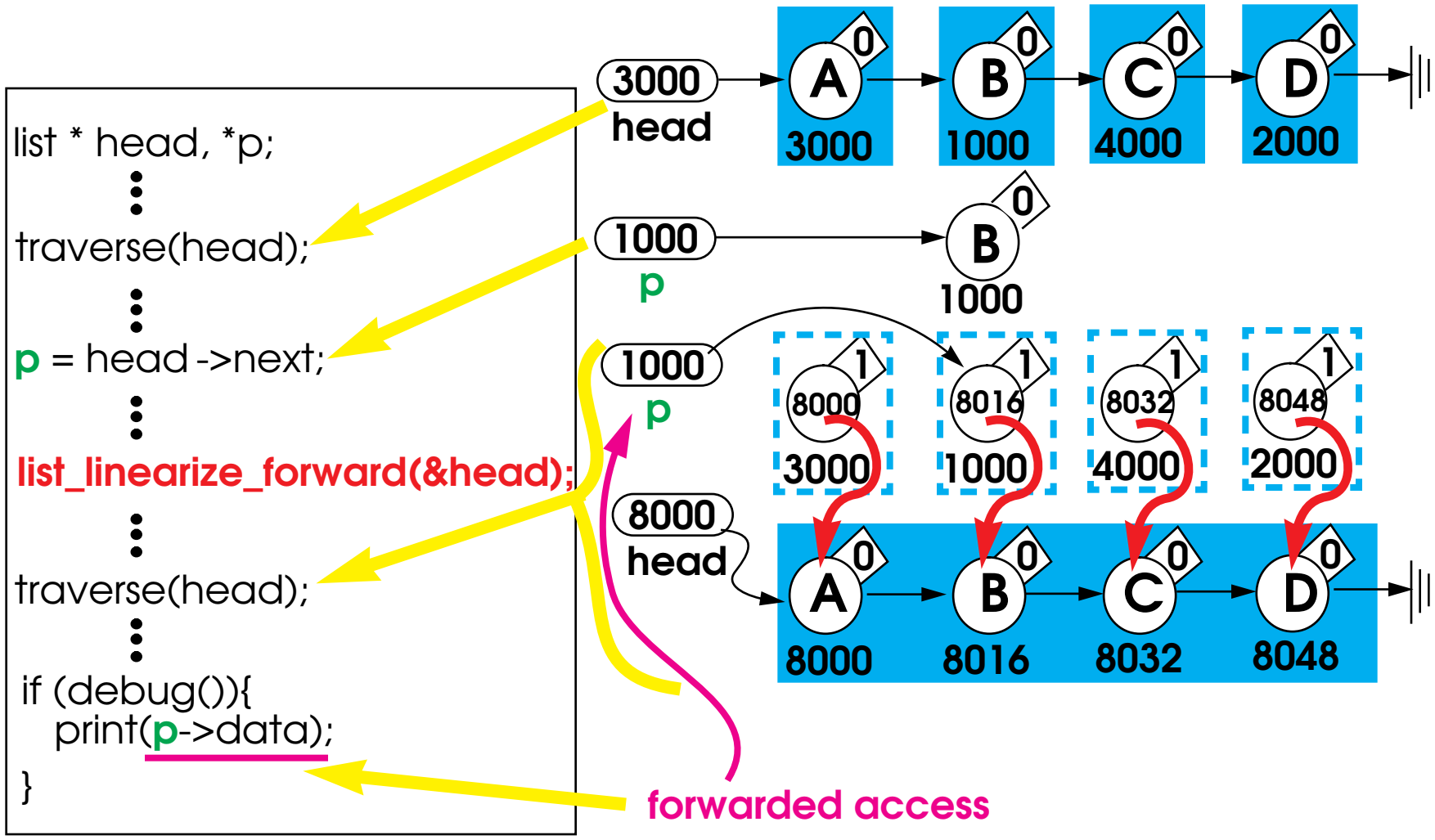
- **Some form of memory forwarding appeared in early Lisp Machines:**
  - e.g., MIT LISP Machine (Greenblatt `74), Symbolic 3600 (Moon `85)

	Memory Forwarding	Garbage Collection
Main Purpose	improve cache performance	conserve memory space
Target Languages	C, C++ *(can't always tell if a data object is "dead")*	Lisp, ML, Java etc. *(can track down all "dead" data objects)*
Function of Forwarding	preserve correctness across the entire execution *(forwarding addresses must be kept around the whole execution)*	maintain data consistency during garbage collection *(forwarding addresses can be discarded after each time garbage is collected)*

☞ **Real-time garbage collectors (Schmidt and Nilsen '94) could benefit from memory forwarding hardware support:**

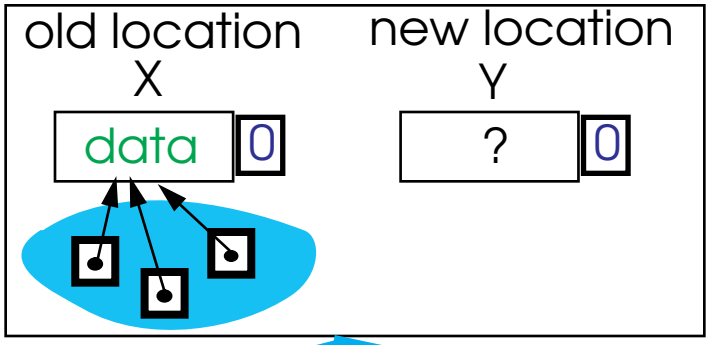
- replace software tags for pointers with our forwarding bits

List Linearization (With Memory Forwarding)



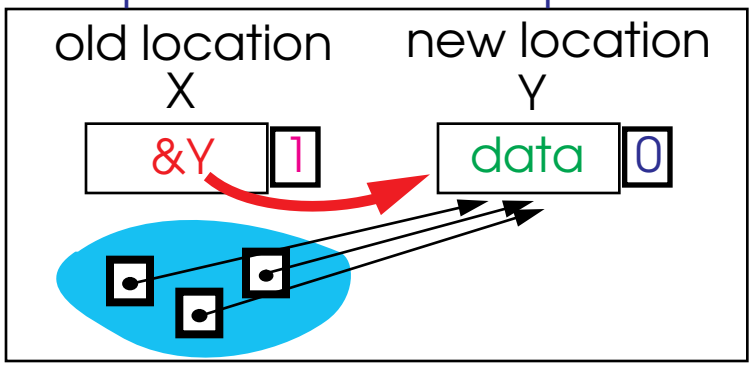
Speculating that Pointers Have Been Updated

Before Relocation:

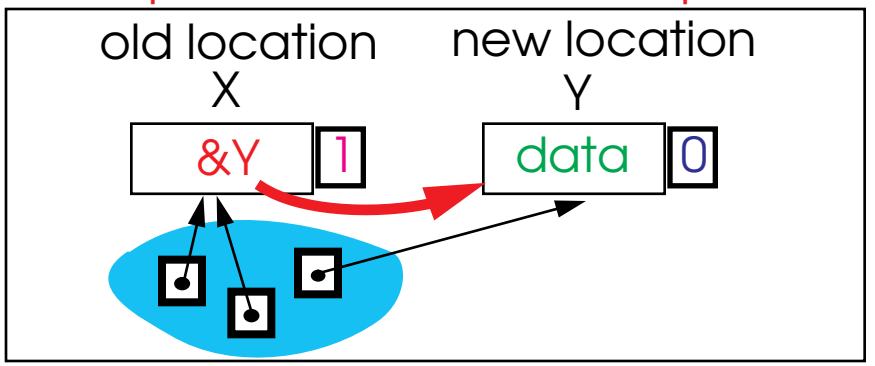


After Relocation:

all pointers to X are updated



some pointers to X are NOT updated



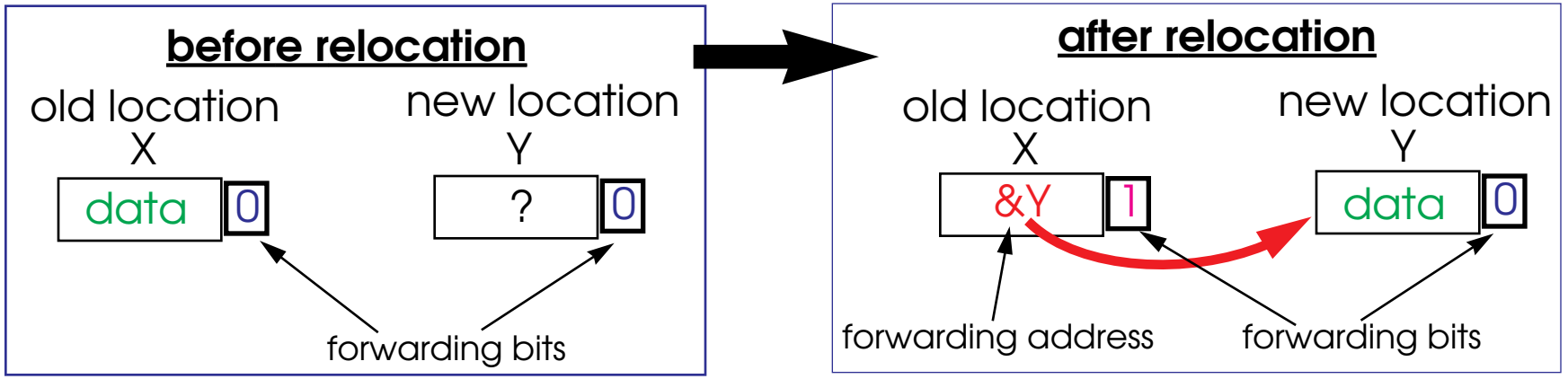
- **Software speculates that all pointers to the old location are updated**
- **If wrong, there is a recovery cost but the execution is still correct**

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

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**

Overview

- ✓ Dynamic Data Layout Optimizations
- ☞ Memory Forwarding: Basic Concepts
- Implementation Issues
- Experimental Evaluation
- Conclusions

Current Approaches to Relocation Safety

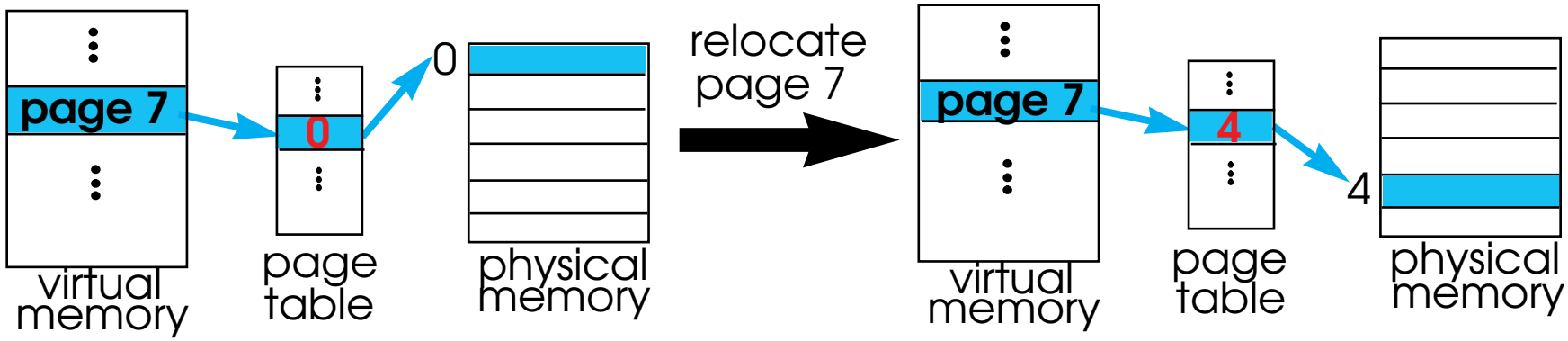
By Programmer:

- error prone for large programs

By Compiler:

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

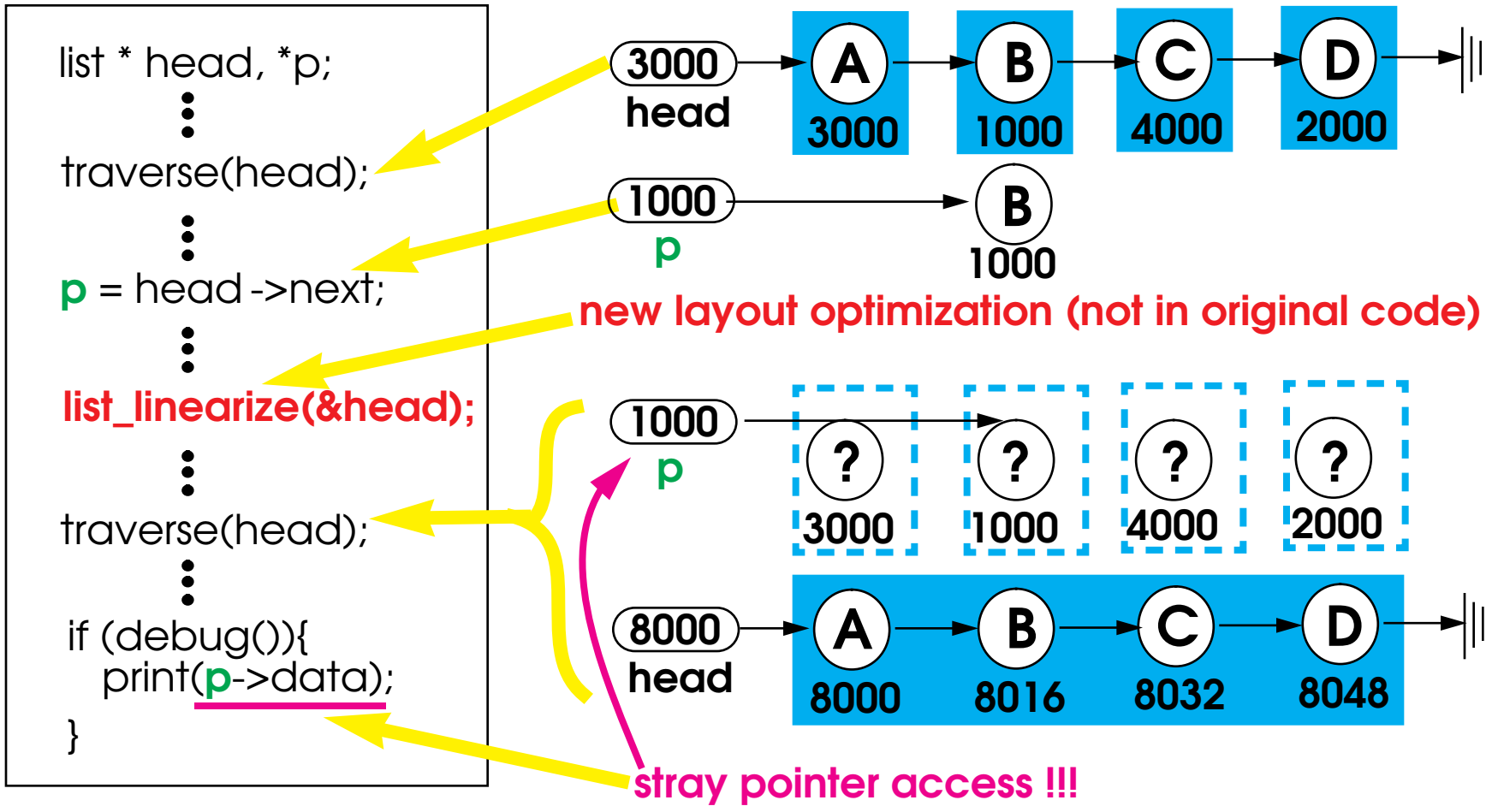
By Virtual Memory System:



- 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***

List Linearization: Example



☞ need techniques to 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

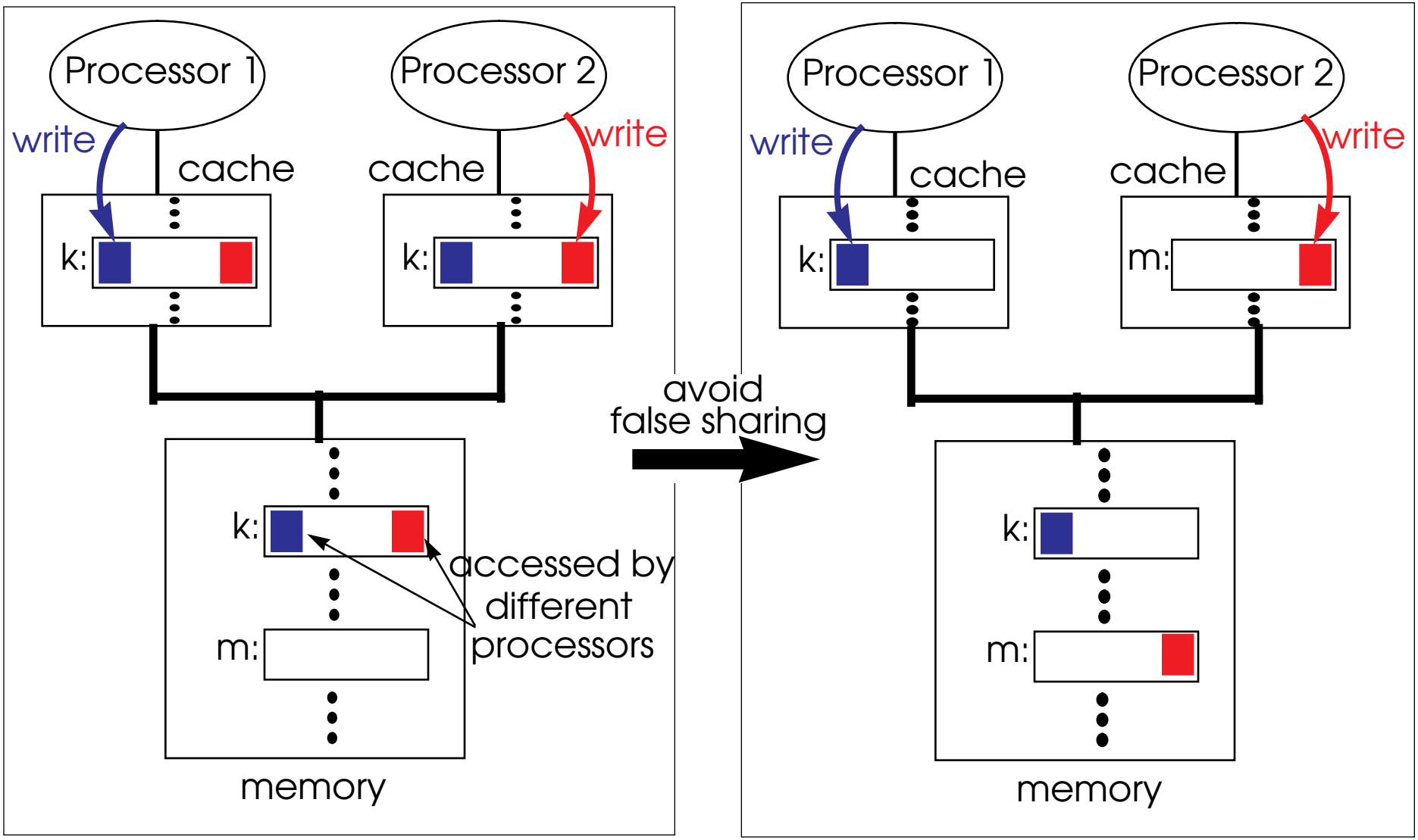
Data-Linearization Prefetching (Luk and Mowry '96):

- map nodes close in the traversal to contiguous memory locations

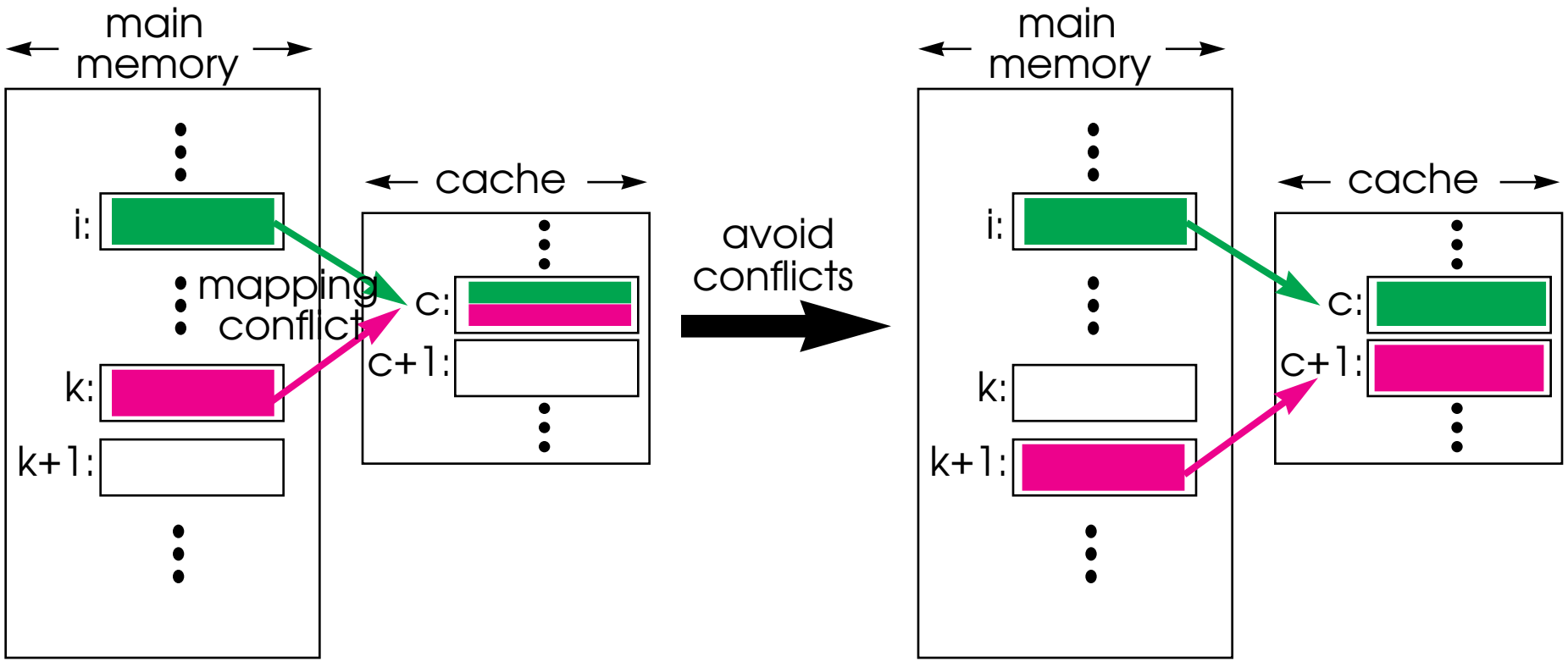
Static vs. Dynamic Layout Optimizations

	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**

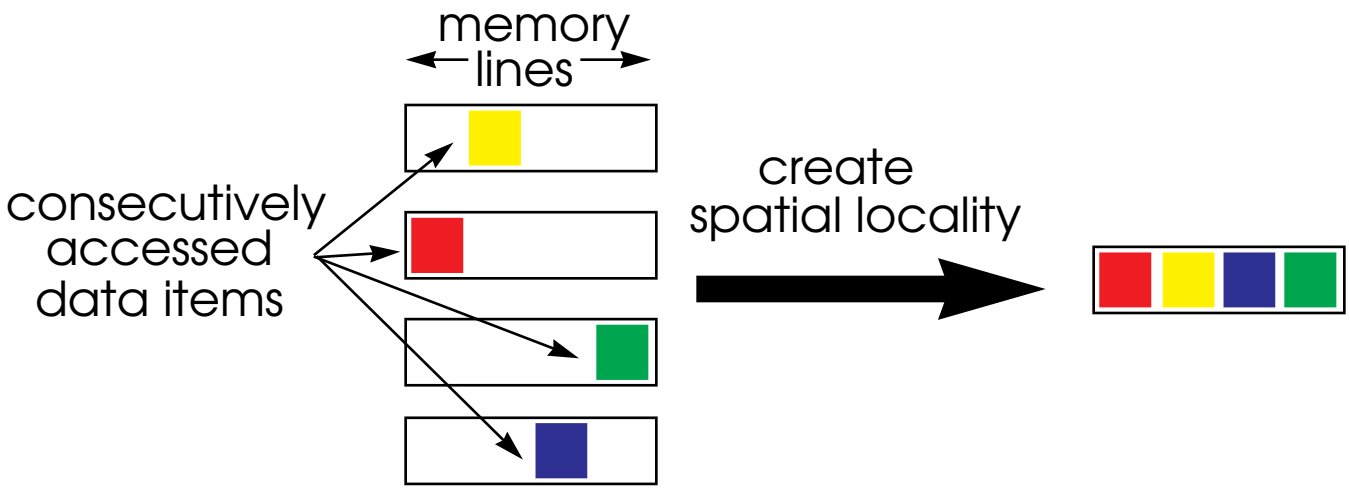
Layout Optimizations: Avoiding False Sharing



Layout Optimizations: Avoiding Cache Conflicts



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**

Data Locality Optimizations

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**

The Importance of Memory Performance

- **Memory subsystem is becoming a key performance bottleneck**
  - latency
  - bandwidth

- **Caches are an important first step, but *not* a complete solution**

  ☞ need to manage caches more intelligently

  ☞ **one approach: data locality optimizations**

Memory Forwarding:

Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation

Chi-Keung Luk

Department of Computer Science

University of Toronto

Todd C. Mowry

Computer Science Department

Carnegie Mellon University