Thesis Proposal on

Optimizing the Cache Performance of Non-Numeric Applications

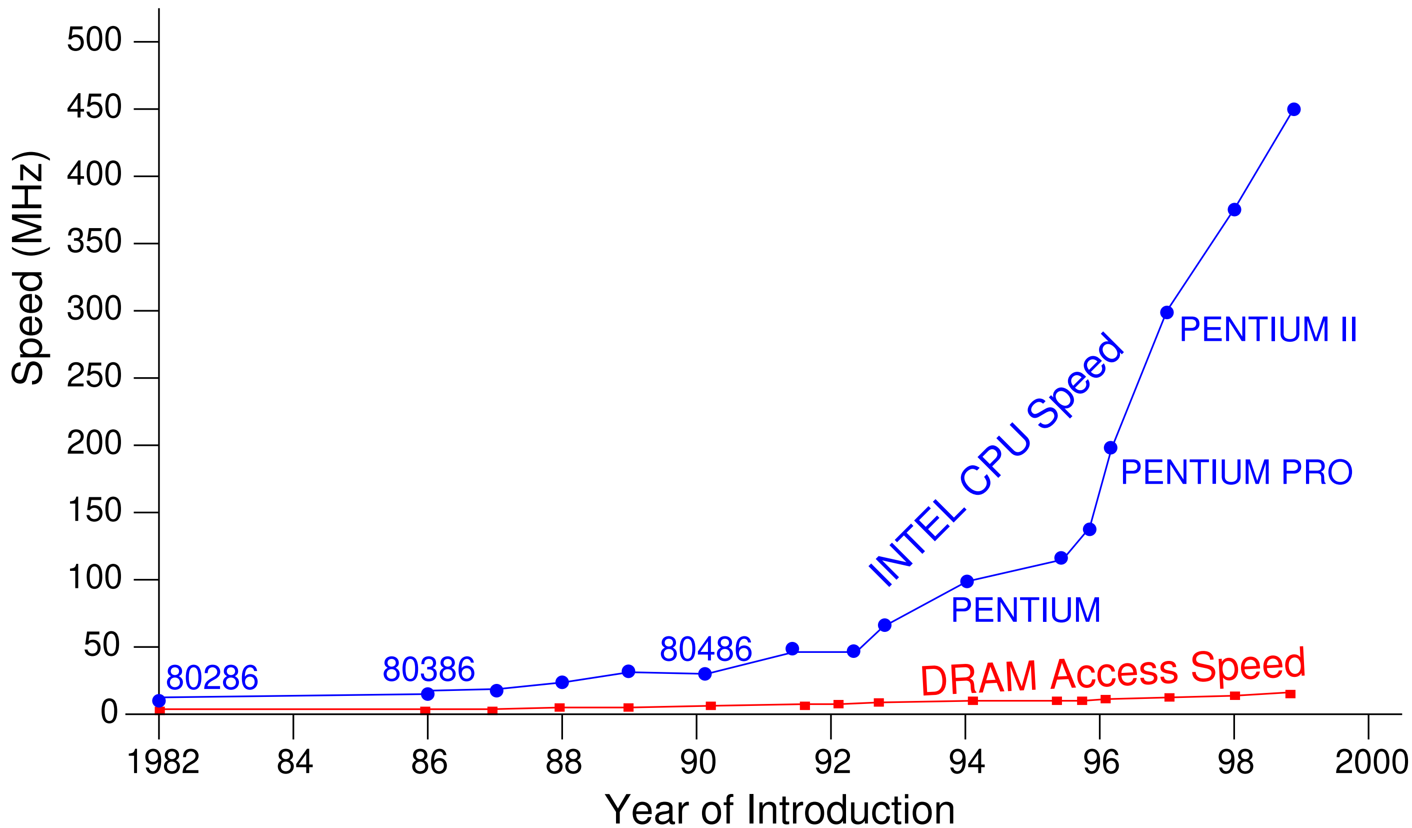
Chi-Keung Luk

Department of Computer Science

University of Toronto

The Memory Latency Problem

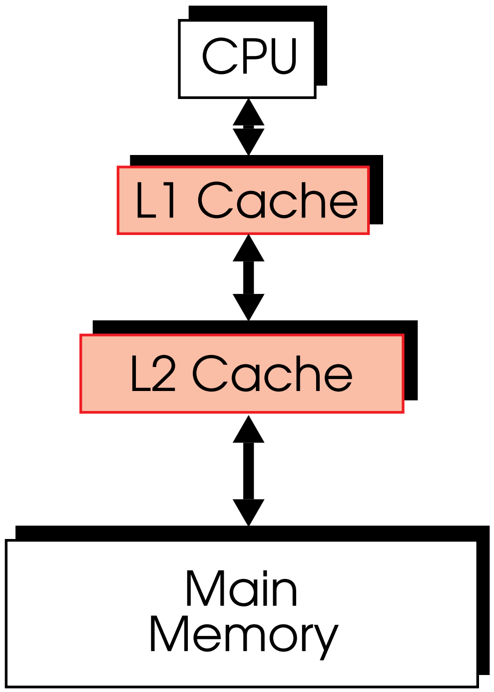
- Memory subsystem supplies both instructions and data
- ↑ processor speed >> ↑ memory speed



☞ **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**
  - size and associativity are constrained by access time
  - insufficient reuse of cached data

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, Personal

☞ 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**
  - techniques that succeed in numeric codes are of little help

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

Overview

☞ **Tolerating Memory Latency**

- **Latency-Hiding Techniques**
- **Our Strategy**

- **Preview of Results**

- **Conclusions**

Latency-Hiding Techniques

Reduce Latency:

- Locality Optimizations
  - restructure computation (only for data)
  - reorganize layout (for both instructions and data)
  - purely software-based

Tolerate Latency:

- Prefetching
  - request instructions/data before they are needed
  - hardware-controlled or software-controlled

- Multithreading
  - switch contexts on long-latency operations
  - purely hardware-based

Challenges with Non-Numeric Codes

Technique	Challenges in Coping with	
	Instruction Latency	Data Latency
Locality Optimizations	1. Need Profiling Information 2. High Sensitivity to Profile Quality	Ensuring Correctness
Prefetching	Scheduling Early Enough	Anticipating Future Data Addresses
Multithreading	1. Finding Concurrency 2. Expensive Hardware Support	

Our Strategy

Instruction Latency:

- tolerate latency through instruction prefetching
  - ☞ techniques to schedule prefetches far ahead of time

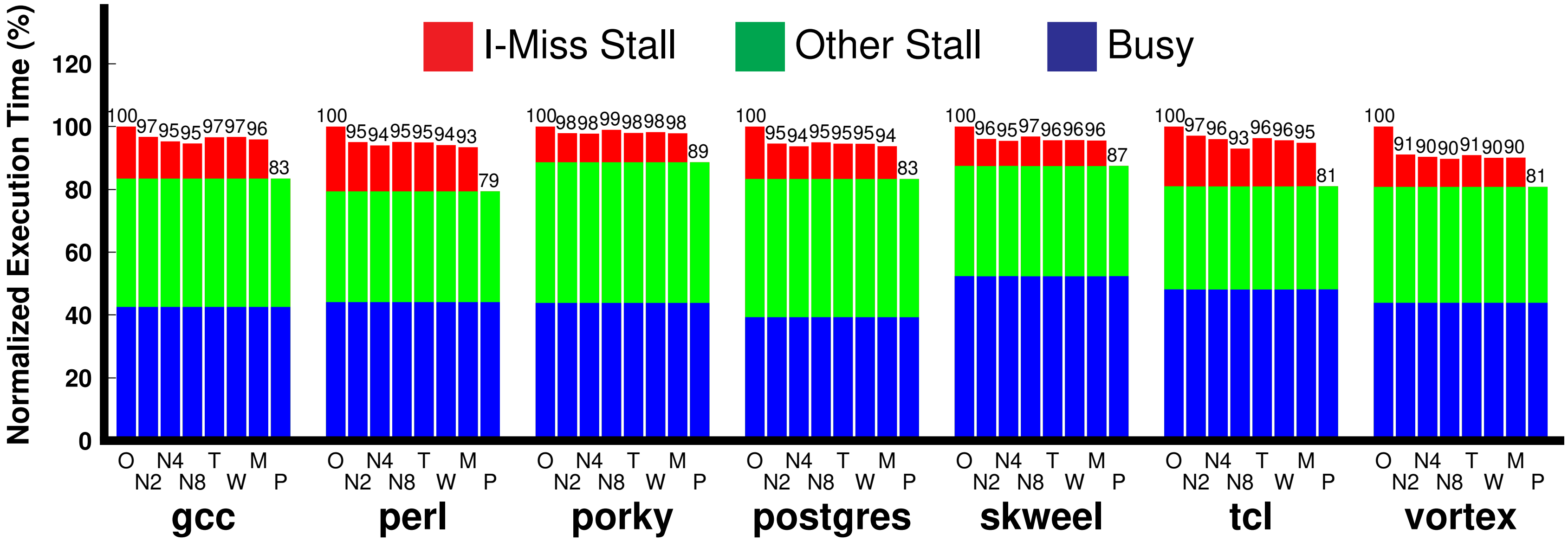
Data Latency: (primary focus is on *pointer-based data structures*)

- tolerate latency through software-controlled prefetching
  - ☞ techniques to predict future data addresses
- minimize overheads of software-based latency tolerance methods
  - ☞ techniques to predict when cache misses happen
- reduce latency using data layout optimizations
  - ☞ techniques to guarantee the safety of these optimizations

Overview

- ✓ Tolerating Memory Latency
- ☞ Preview of Results
  - ☞ Cooperative Instruction Prefetching
  - Compiler-Based Data Prefetching
  - Correlation-Based Miss Prediction
  - Memory Forwarding
- Conclusions

Performance of State-of-the-Art Instruction Prefetching



O = Original | **N*x*** = Next-*x*-line prefetching | **T** = Target-line prefetching
W = Wrong-path prefetching | **M** = Markov prefetching | **P** = Perfect I-cache

All schemes offer similarly unsatisfactory performance:

- only 1/3 of I-miss stall time is hidden
- little (or no) performance gain from non-sequential prefetching

Problem: Prefetches are launched too late!

☞ **Our solution:** ***Cooperative Instruction Prefetching***

Cooperative Instruction Prefetching

Goals:

- prefetch all original I-cache misses
  - while not polluting the cache (avoid *useless prefetches*)
- prefetch far enough ahead to fully hide latency

Our Approach:

Hardware:

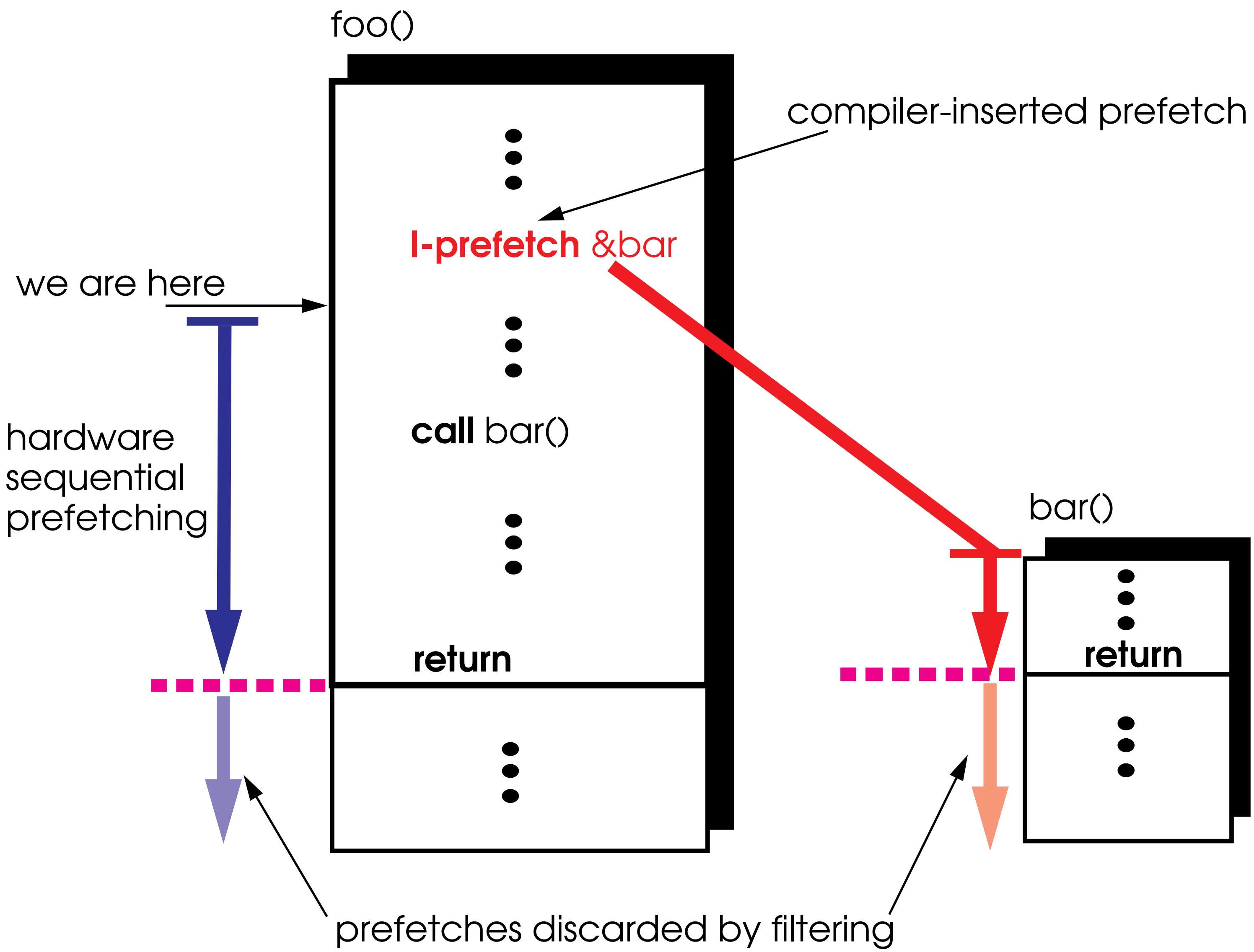
- prefetch filter to eliminate useless prefetches (avoids pollution)
- aggressive sequential prefetching (improves timeliness)

Compiler:

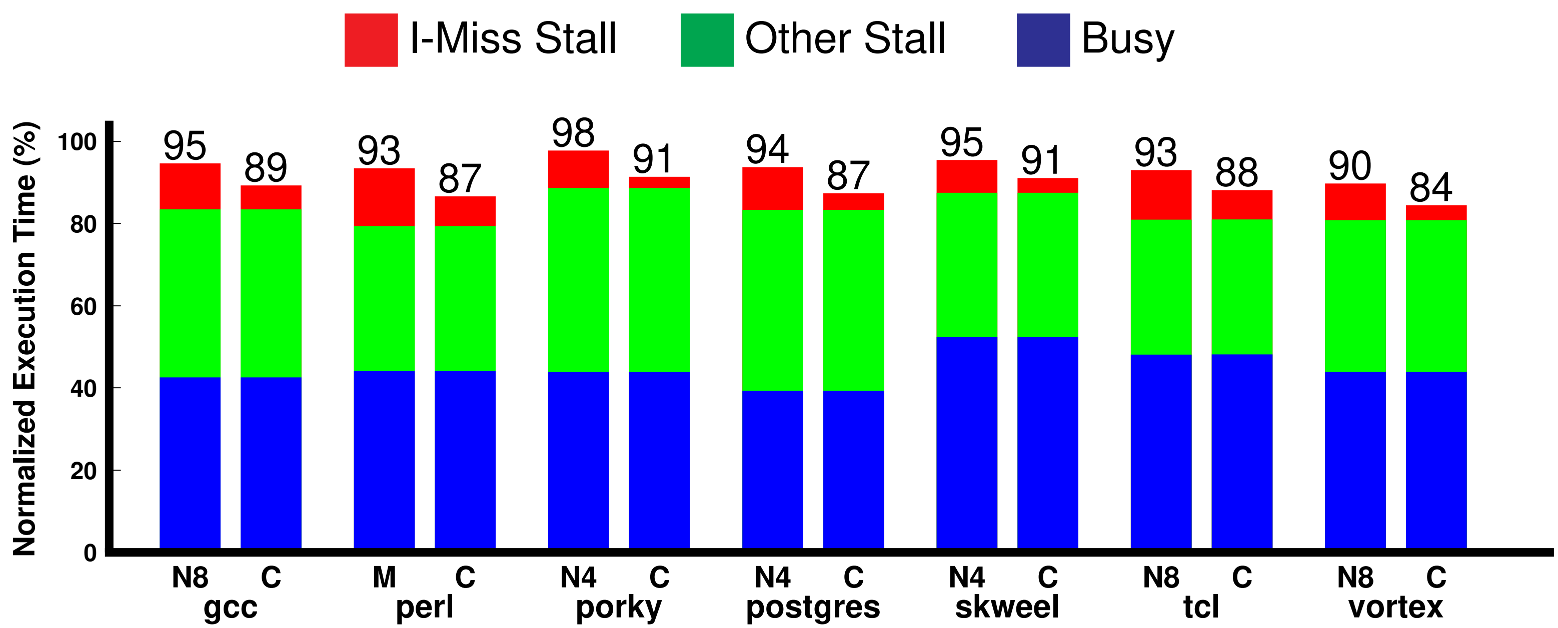
- schedules non-sequential prefetches early enough

☞ **A cooperative approach using both hardware & compiler support**

Instruction Prefetching Strategies



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
  - ☞ **6.4% speedup over best previous schemes**
  - ☞ **13.3% speedup over no prefetching**

Overview

- ✓ Tolerating Memory Latency
- ☞ **Preview of Results**
  - ✓ Cooperative Instruction Prefetching
  - ☞ **Compiler-Based Data Prefetching**
  - Correlation-Based Miss Prediction
  - Memory Forwarding
- Conclusions

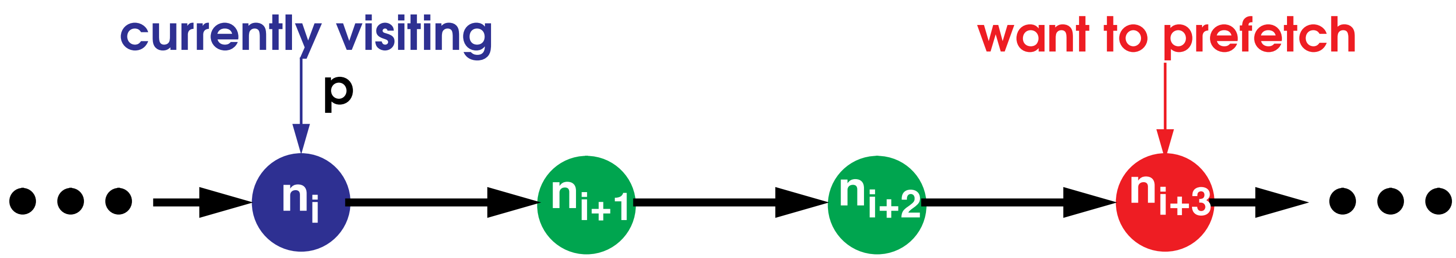
Recursive Data Structures

- **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

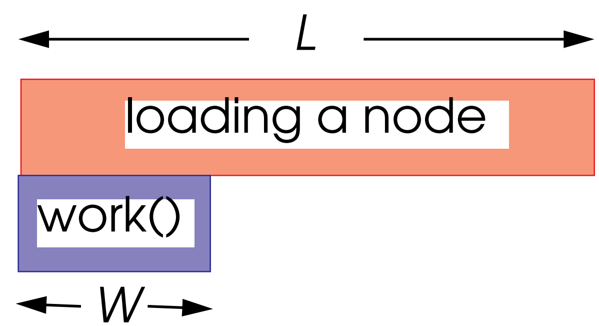
Goal:

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

Scheduling Prefetches for Recursive Data Structures




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



Our Goal: *fully hide latency*

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

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

The Pointer-Chasing Problem

No Prefetching
Comp. Rate = $1/(L+W)$

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

Prefetch 1 Node Ahead
Comp. Rate = $1/L$

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

Prefetch 3 Nodes Ahead (Naively)
Comp. Rate Still = $1/L$

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

Ideal Prefetching
Comp. Rate = $1/W$

```
while (p){
    pf(&n_{i+3});
    work(p->data);
    p = p->next;
}
```

Pointer-Chasing Problem:

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

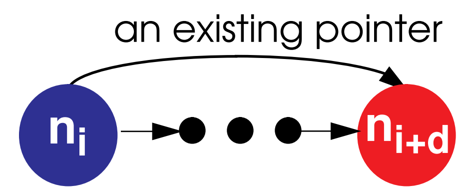
Overcoming the Pointer-Chasing Problem

Key:

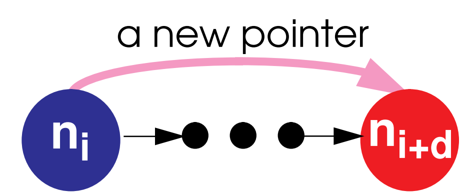
☞ **n_i needs to know $\&n_{i+d}$ without referencing the d-1 intermediate nodes**

Three approaches:

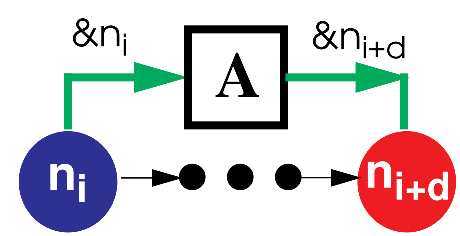
- use *existing* pointer(s) in n_i to approximate $\&n_{i+d}$
  - e.g. **Greedy Prefetching**



- add *new* pointer(s) to n_i to approximate $\&n_{i+d}$
  - e.g. **History-Pointer Prefetching**



- compute $\&n_{i+d}$ *directly* from $\&n_i$ (no ptr. deref.)
  - e.g. **Data-Linearization Prefetching**

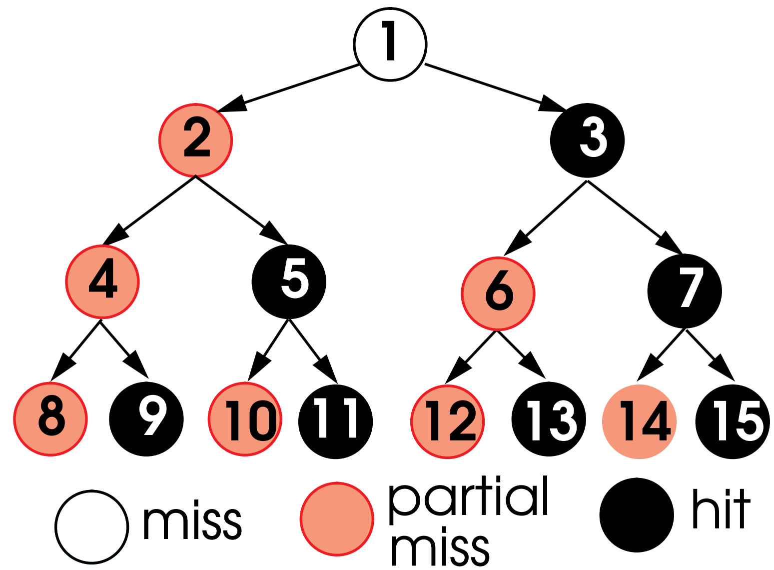


A=Address generating function

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 and widely applicable**

  ☞ **fully implemented in SUIF**

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

More Sophisticated Prefetching

History-Pointer Prefetching:

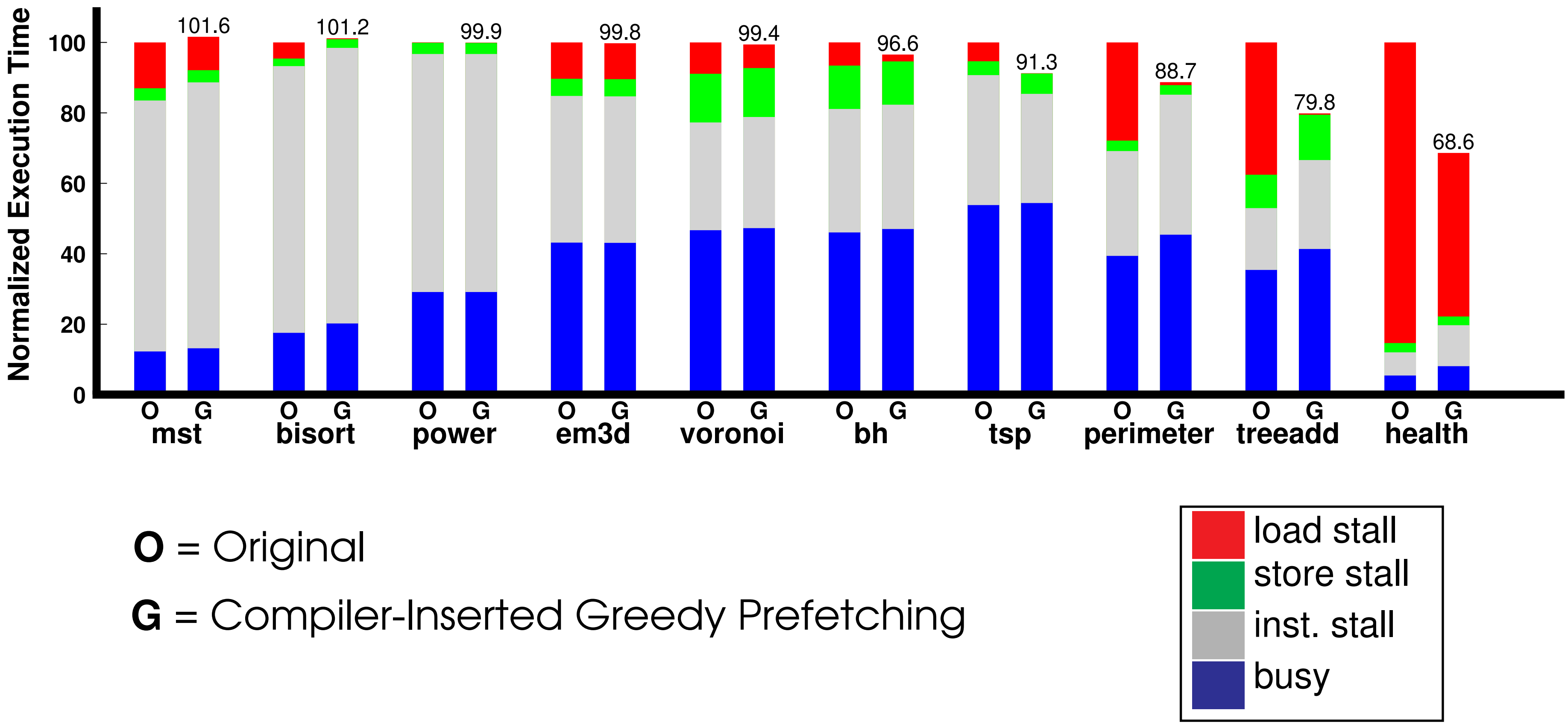
- add new pointer(s) called history-pointers to each node
- history-pointers are obtained from some recent traversal
  - ☞ 1 dereference to approximate $\&n_{i+d}$ at n_i

Data-Linearization Prefetching:

- map nodes close in the traversal to contiguous memory
  - ☞ 0 dereference to approximate $\&n_{i+d}$ at n_i

☞ **trade space & time for better control over prefetching distances**

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%**

Overview

✓ Tolerating Memory Latency

☞ Preview of Results

  ✓ Cooperative Instruction Prefetching

  ✓ Compiler-Based Data Prefetching

  ☞ Correlation-Based Miss Prediction

  • Memory Forwarding

• Conclusions

Software-Based Latency Tolerance Techniques

Examples:

- prefetching, scheduling non-blocking loads early, etc.

Advantages:

- exploit knowledge of application-wide behavior
- flexible
- minimal hardware support

Key Challenge:

- maximizing gain while minimizing overheads

  ☞ ***apply only to dynamic references that suffer misses***

Predicting Data Cache Misses

Numeric Applications:

- static locality analysis works reasonably well (Mowry *et al*, 1992)
- strip-mining, peeling, etc. can be used to isolate miss instances

Non-Numeric Applications:

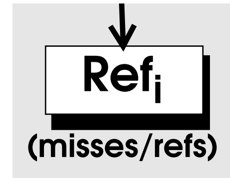
- static locality analysis is extremely **difficult**:
  - pointers, complex control flow, etc.

☞ one possibility: ***use profiling information***

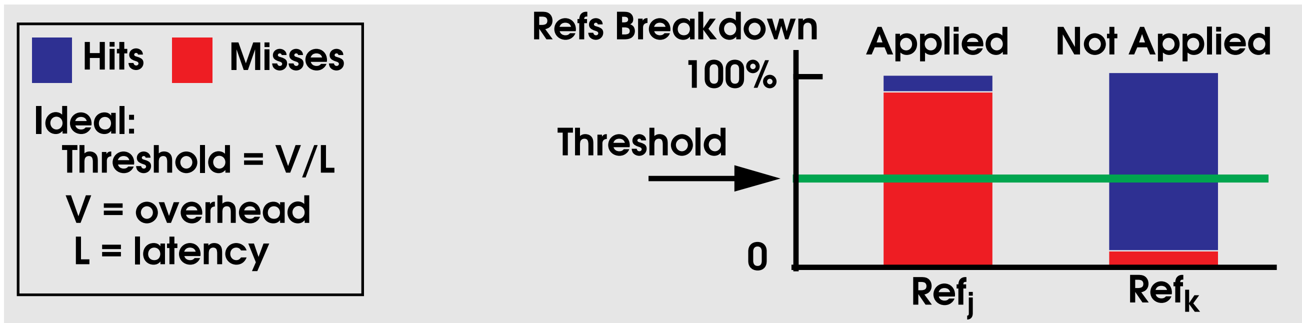
State-of-the-Art: Summary Profiling

Information Collected:

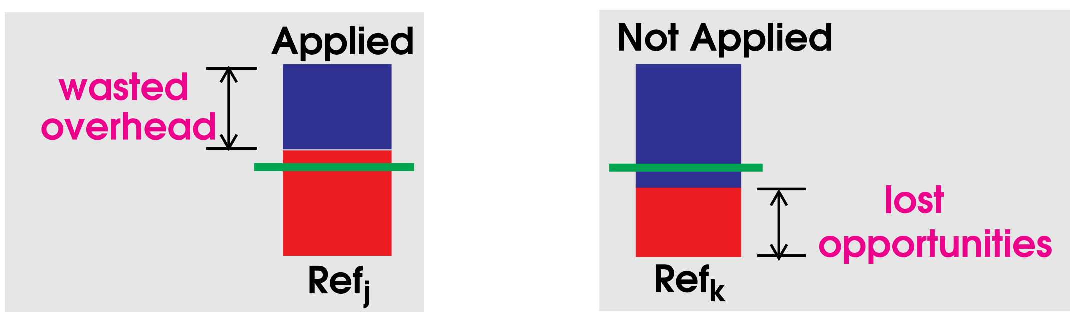
- precise miss ratio of each memory reference



Latency Tolerance Strategy: ***all-or-nothing* for a given reference**



The Problem: **references with intermediate miss ratios**



Ideally: precisely isolate dynamic miss instances

☞ ***our solution: Correlation Profiling***

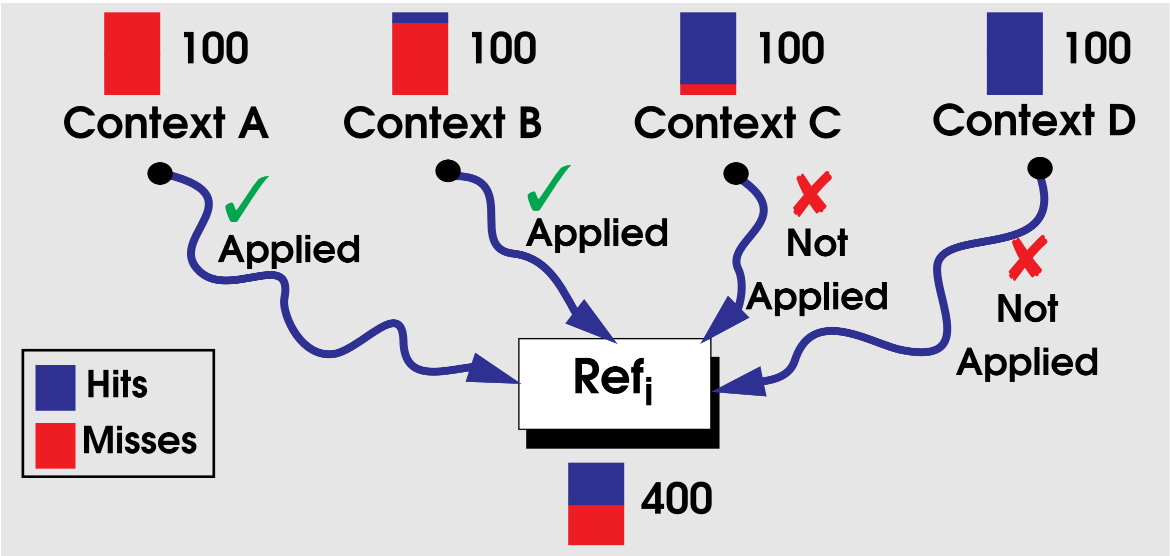
Correlation Profiling

Information Collected:

- precise miss ratios of dynamic **contexts**
  - context: instances of a ref. with a common distinguishable pattern

Latency Tolerance Strategy:

- contexts are statically isolated in the code
- decision is made on a context-by-context basis



☞ **enables further refinement of the optimization strategy**

Correlation Profiling Techniques

Control-Flow Correlation:

- correlate cache outcomes with recent control-flow paths
- useful for detecting data reuse or cache displacement

Self Correlation:

- correlate cache outcomes with previous instances of the same load
- useful for detecting:
  - spatial locality non-obvious to compilers
  - sequences of all hits, all misses

Global Correlation:

- correlate cache outcomes with previous instances of the any load
- useful for detecting bursty patterns across multiple loads

Impact on Prediction Accuracy: Eqntott

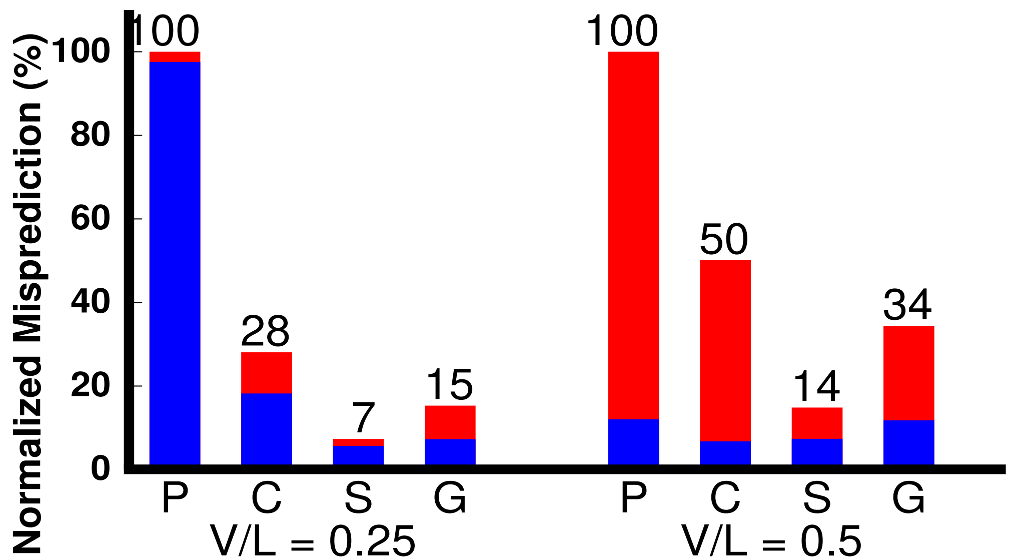
Predict HIT / Actual MISS
(lost opportunities)

Predict MISS / Actual HIT
(wasted overhead)

P = summary   C = control-flow
S = self   G = global

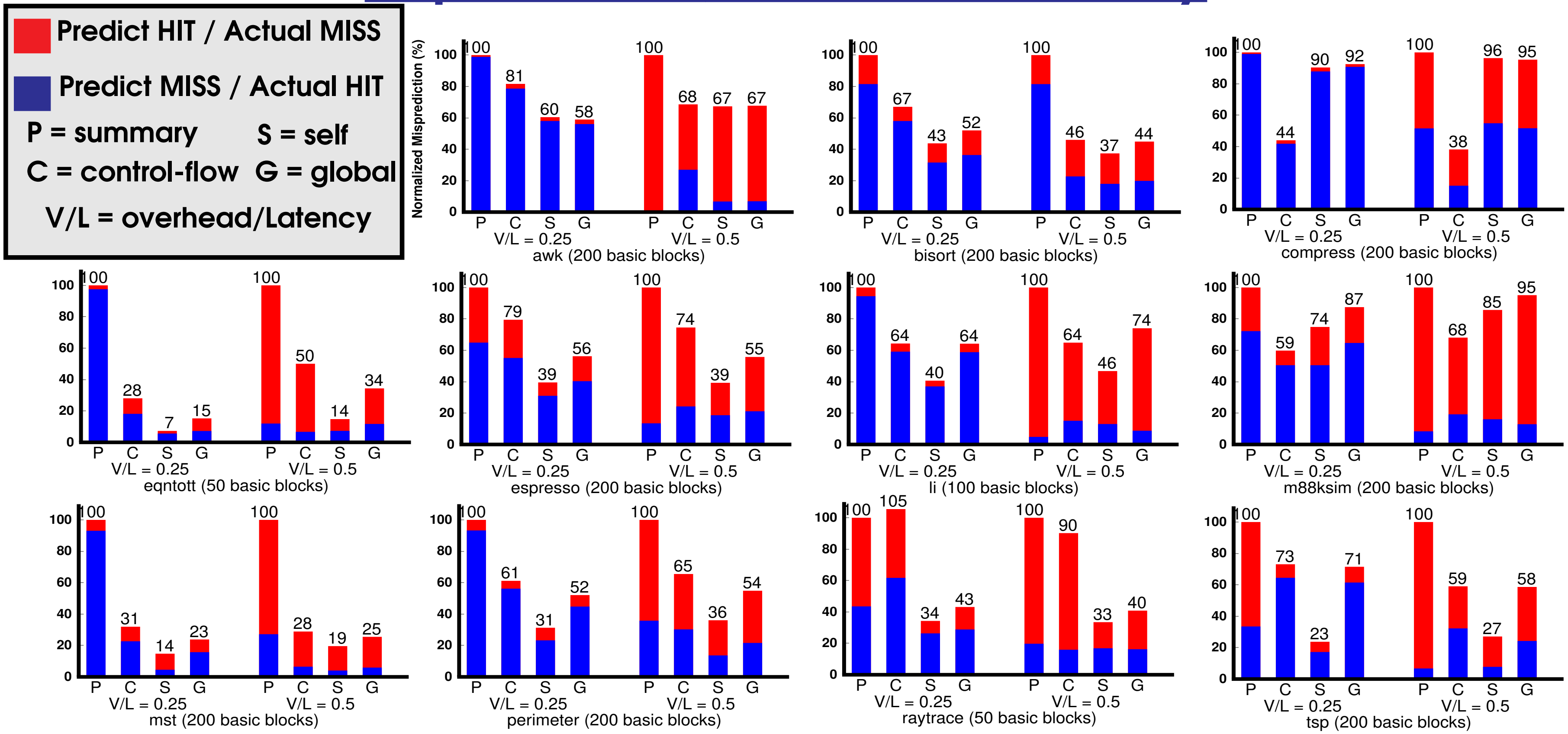
V / L = overhead to latency ratio

Misprediction Count



☞ *all three correlation profiling schemes improve prediction accuracy*

Impact on Prediction Accuracy



- *All three* schemes offer significant improvements in prediction accuracy
- **Self:** recognizes (i) non-obvious spatial locality, (ii) runs of all hits / misses
- **Global:** essentially assimilates self correlation
- **Control-flow:** captures call-chain dependent behaviors

Overview

✓ **Tolerating Memory Latency**

☞ **Preview of Results**

- ✓ **Cooperative Instruction Prefetching**
- ✓ **Compiler-Based Data Prefetching**
- ✓ **Correlation-Based Miss Prediction**
- ☞ **Memory Forwarding**

- **Conclusions**

Data Layout Optimizations

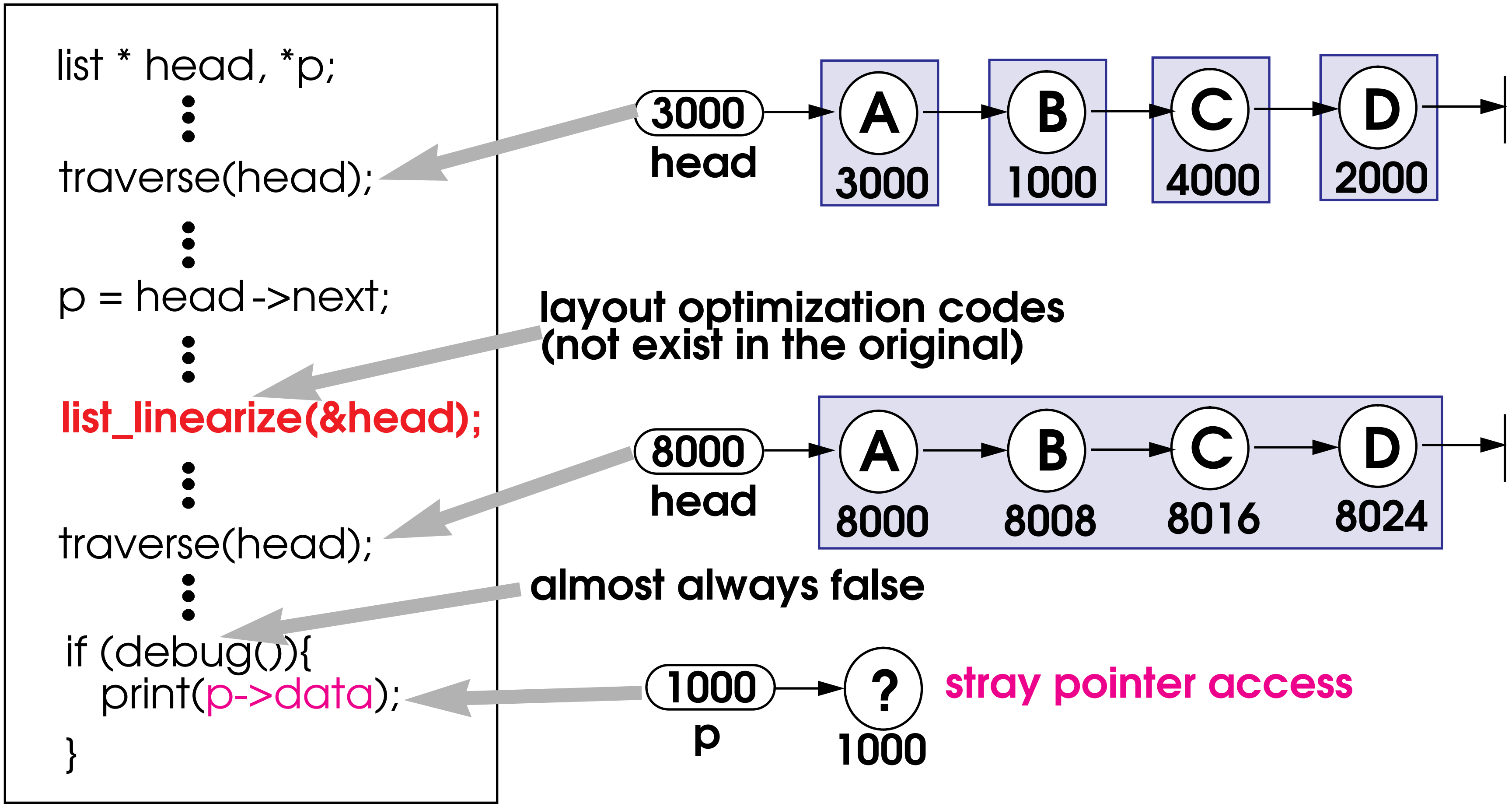
- **Reorganize data layouts to *reduce latency* via:**
  - creating spatial locality
  - avoiding cache conflicts
  - avoiding false sharing

- **Static vs. dynamic layout optimizations:**
  - static: fix the ``optimized'' data addresses at object creation time
    - relatively easy to implement
  - dynamic: relocate data objects across entire execution
    - adapt to varying program behaviors
    - impressive performance potential

Key Challenge of Dynamic Layout Optimizations:

- guarantee the safety of data relocation

Example: List Linearization



Problem: Software is forced to sacrifice performance for safety

☞ ***our solution:*** ***Memory Forwarding***

Memory Forwarding

Basic Ideas:

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

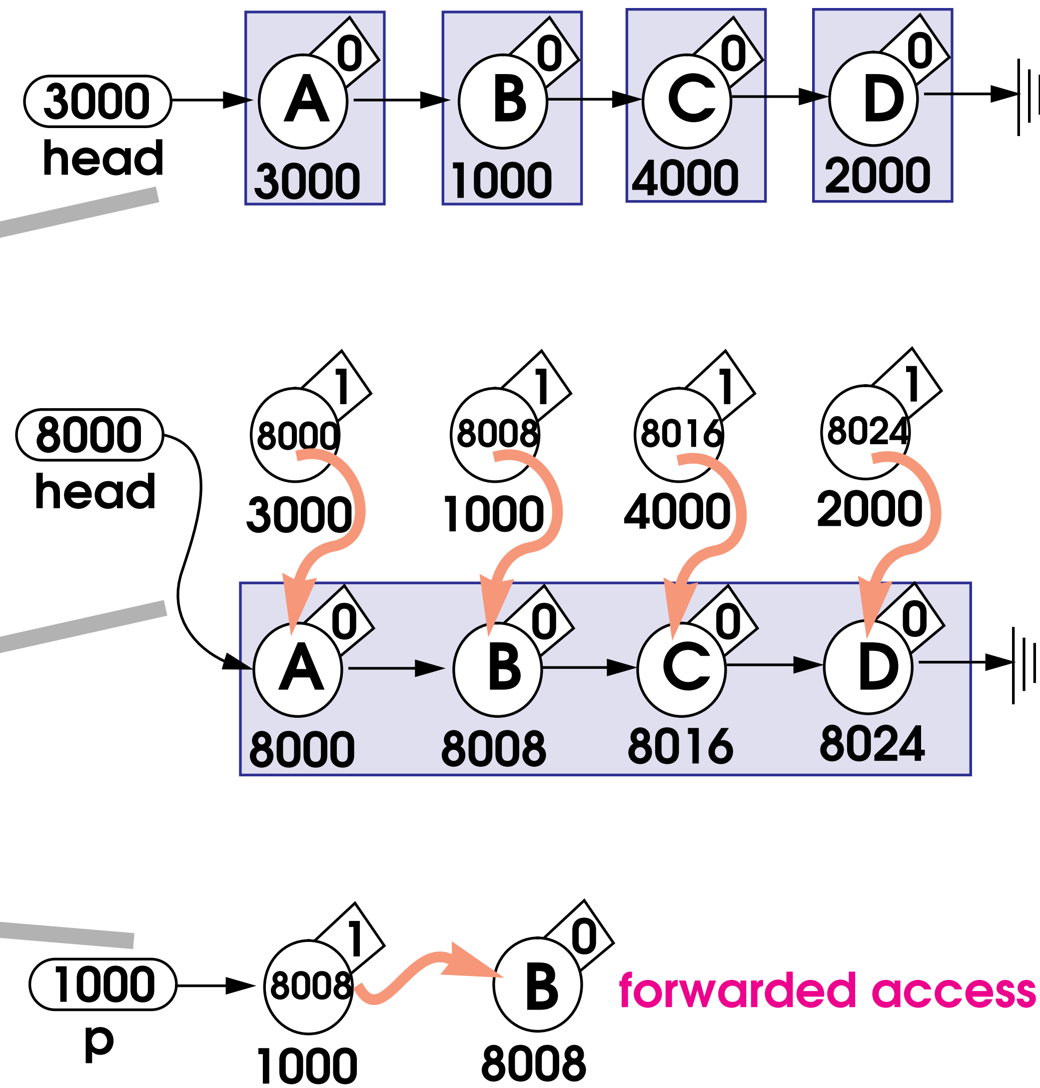
- while relocating an object:
  - store its new address into its old location
  - mark the old location as containing a forwarding address
- whenever the old location is accessed:
  - hardware automatically forwards the access to the new location

Advantages:

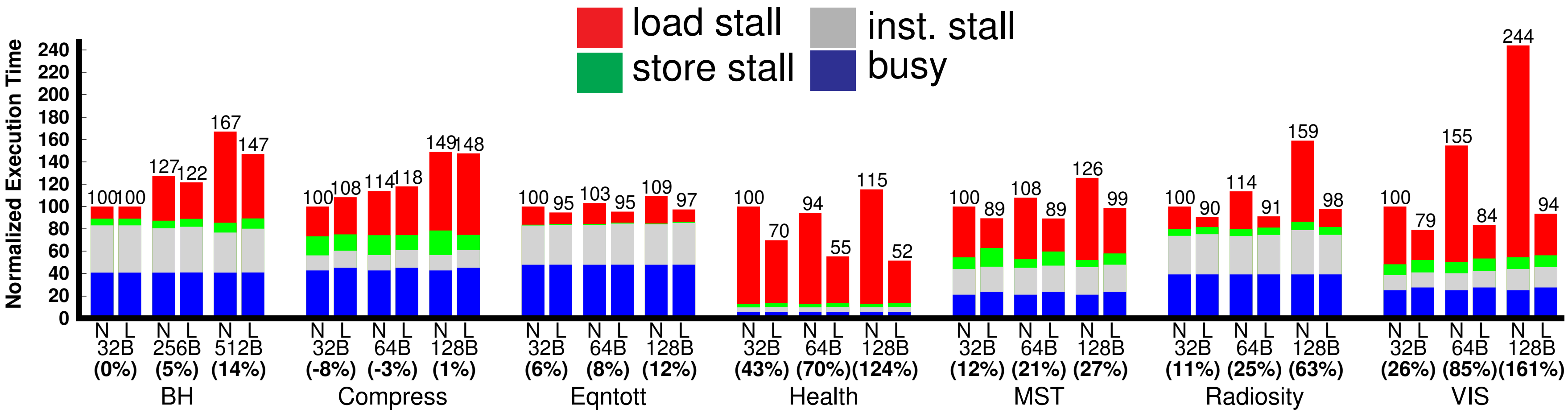
- guarantee the safety of *any* kinds of data relocation
- support fine-grain relocation (at a *word* granularity)
- low overheads:
  - space: 1 bit/word (1.5% for 64-bit words)
  - run-time: paid *only if* forwarding actually occurs

List Linearization (With Memory Forwarding)

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



Performance of Aggressive Layout Optimizations



N = Not Optimized	***x*B** = with *x*-byte cache lines
L = Layout Optimized	**(*y*%)** = *y*% speedup of **L** over **N**

- **Lack of spatial locality in the unoptimized cases**

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

☞ **achieve significant speedups in many cases**

Overview

✓ **Tolerating Memory Latency**

✓ **Preview of Results**

☞ **Conclusions**

- ☞ **Contributions**
- **Plan**

Primary Contributions of Dissertation

- **Effective instruction prefetching techniques for modern processors**
  - prefetch early enough *and* eliminate useless prefetches
    - ☞ reduce much of the latency remaining with state-of-the-art

- **Unique compile-based techniques for prefetching pointer-based codes**
  - fully implemented greedy prefetching in SUIF
    - ☞ automatically improve execution speed significantly

- **Novel profiling techniques for predicting cache misses**
  - cache miss behaviors are strongly correlated to dynamic contexts
    - ☞ potentially reduce overheads of latency tolerance techniques

- **General architectural technique to guarantee safety of data relocation**
  - add a hardware-aware layer of indirection within the memory system
    - ☞ enable many aggressive data layout optimizations

Plan for Completing Dissertation

☞ ***already obtained most of the results, some minor studies left:***

- **Further reduce instruction prefetching overhead using profiling**

- **Study the interaction of instruction locality optimizations and prefetching**

- **Study a few more applications that can benefit from memory forwarding**