Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling

Todd C. Mowry
Computer Science Department
Carnegie Mellon University

Chi-Keung Luk
Department of Computer Science
University of Toronto

Software-Based Latency Tolerance Techniques

Promising approaches for addressing the memory latency problem

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***

Carnegie Mellon Toronto

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:

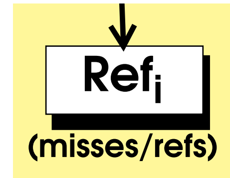
- static locality analysis is extremely **difficult**:
  - pointers, complex control flow, etc.

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

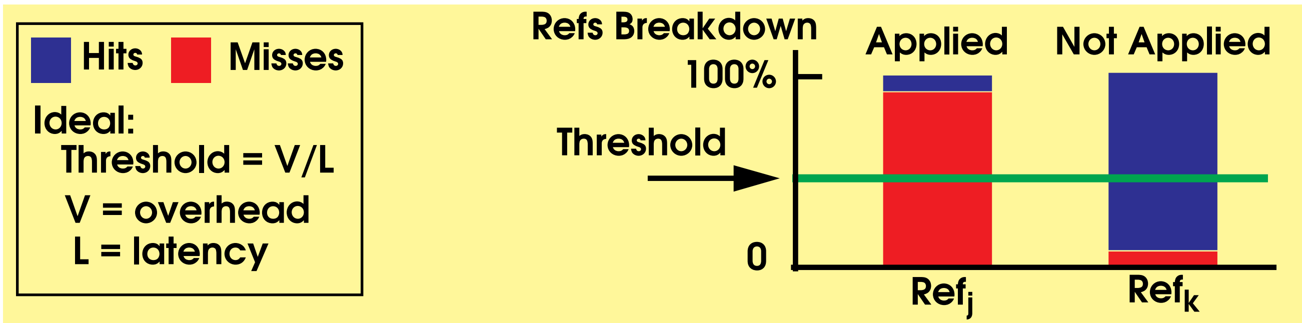
State-of-the-Art: Summary Profiling

Information Collected:

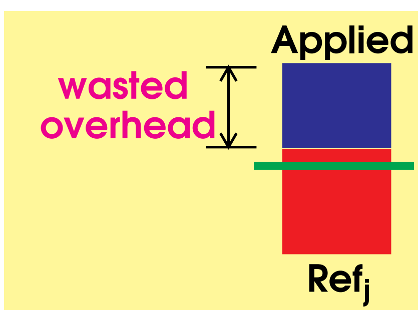
- precise miss ratio of each memory reference

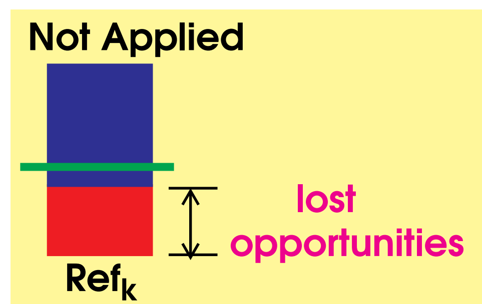


Latency Tolerance Strategy: *all-or-nothing* for a given reference



The Problem: references with intermediate miss ratios





Ideally: precisely isolate dynamic miss instances

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

Carnegie Mellon Toronto

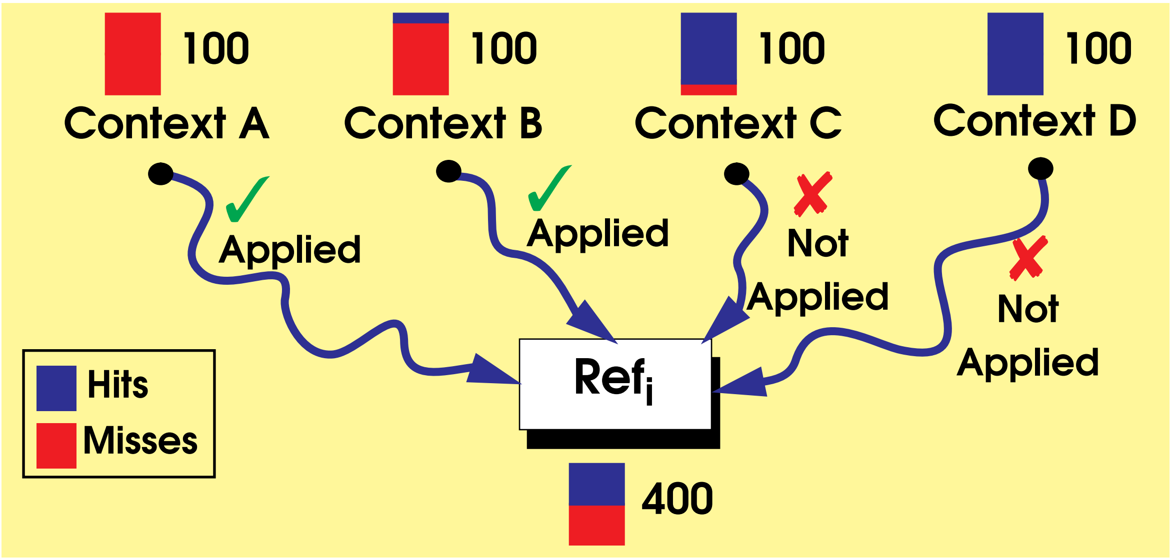
Correlation Profiling

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**

Carnegie Mellon Toronto

Goals

- Can correlation profiling offer improved prediction accuracy?
- Can we translate this improved accuracy into performance gains?
- When and why does correlation profiling succeed?

☞ *we must be able to statically isolate contexts in the source code*

Overview

☞ Correlation Profiling Techniques

- Control-Flow Correlation
- Self Correlation
- Global Correlation

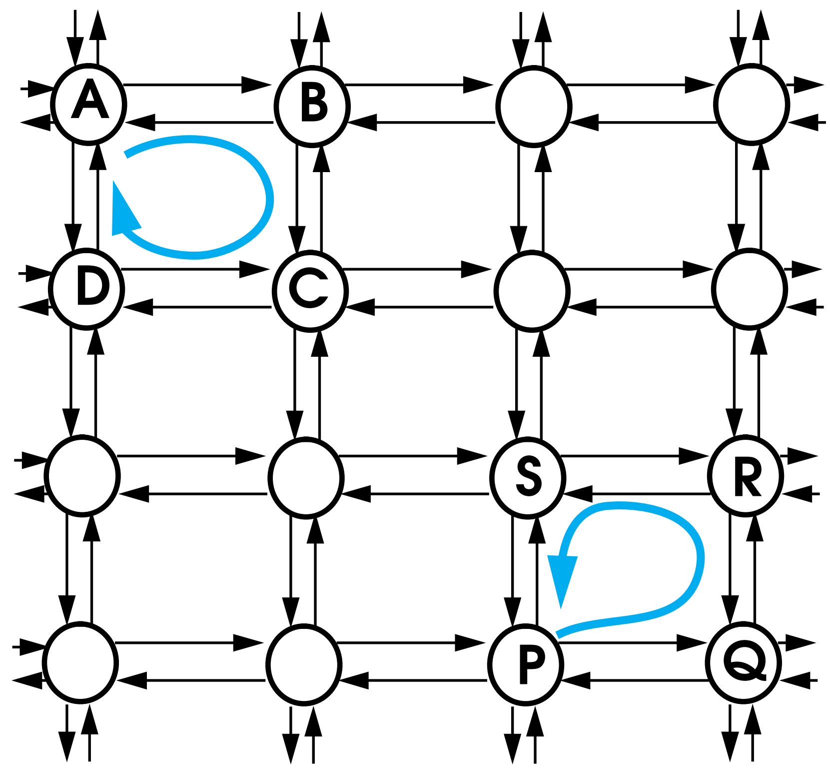
- Experimental Results
- Exploiting Correlation Profiling in Practice
- Conclusions

Control-Flow Correlation

- Correlate cache outcomes with recent control-flow paths
- Useful for detecting data reuse or cache displacement

```
void walk(node* p){
  int dt = p->data;
  work(dt);
  if (go_left(dt))
    p = p->left;
  elsif (go_right(dt))
    p = p->right;
  elsif (go_up(dt))
    p = p->up;
  elsif (go_down(dt))
    p = p->down;
  else p = NULL;
  if (p != NULL) walk(p);
  else return;
}
```

control-flow correlated

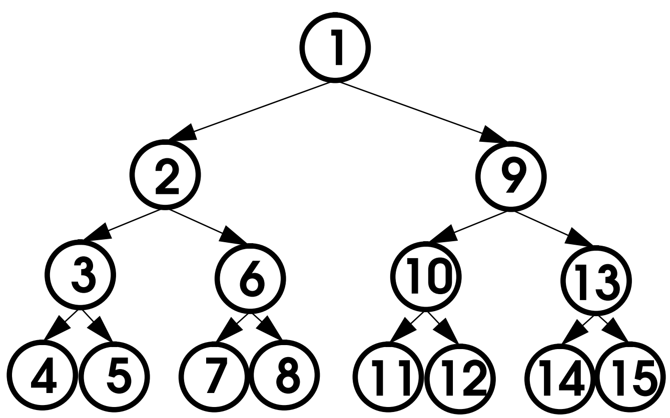


Self Correlation

- **Correlate cache outcome with N previous instances of the same load**
- **Useful for detecting:**
  - spatial locality non-obvious to compilers
  - runs of all hits, all misses

```
preorder(treeNode* p) {
  if (p != NULL) {
    work(p->data);
    preorder(p->left);
    preorder(p->right);
  }
}
```

self correlated



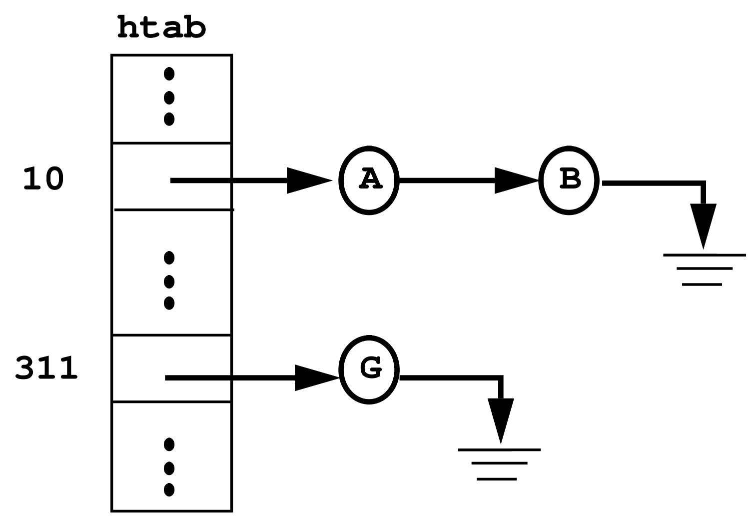
preorder traversal	self cache outcomes
1	M
2	M
3	H
4	M
5	H
6	M
7	H
8	M
9	H
10	M
11	H
12	M
13	H
14	M
15	H

Global Correlation

- Correlate cache outcome with N previous instances of *any* loads
- Useful for detecting bursty patterns across multiple loads

```
while (1) {
  int i = hash(get());
  listNode* curr = htab[i];
  while (curr != NULL){
    work(curr->data);
    curr = curr->next;
  }
}
```

globally correlated



global memory accesses	global cache outcomes
htab[10]	M
A->data	M
A->next	H
B->data	M
B->next	H
...	...
htab[311]	M
G->data	M
G->next	H
...	...
htab[10]	H
A->data	H
A->next	H
B->data	H
B->next	H

Overview

- ✓ Correlation Profiling Techniques
- ☞ **Experimental Results**
  - **Framework**
  - **Case Study: li**
  - **Overall Results**
- **Exploiting Correlation Profiling in Practice**
- **Conclusions**

Experimental Framework

Benchmarks (22 in total):

- **SPECint**: all of 95, remaining from 92
- **SPLASH-2**: uniprocessor versions of raytrace, radiosity
- **Olden**: 8 pointer-intensive applications
- **UNIX utilities**: awk

Compiler:

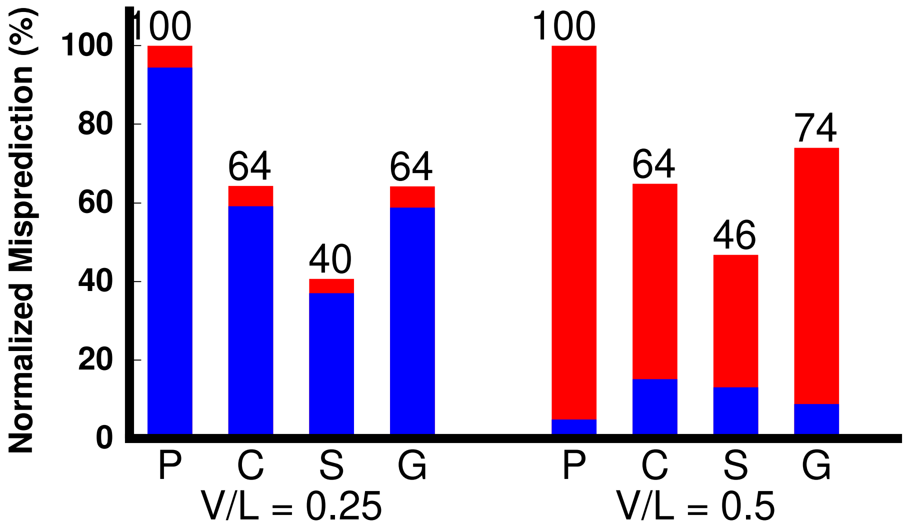
- MIPS C compilers with -O2 under Irix 5.3

Simulator:

- driven by full pixie traces
- simplified version of CELLO
  - models a perfectly-pipelined single-issue processor

Case Study: li

Misprediction Count



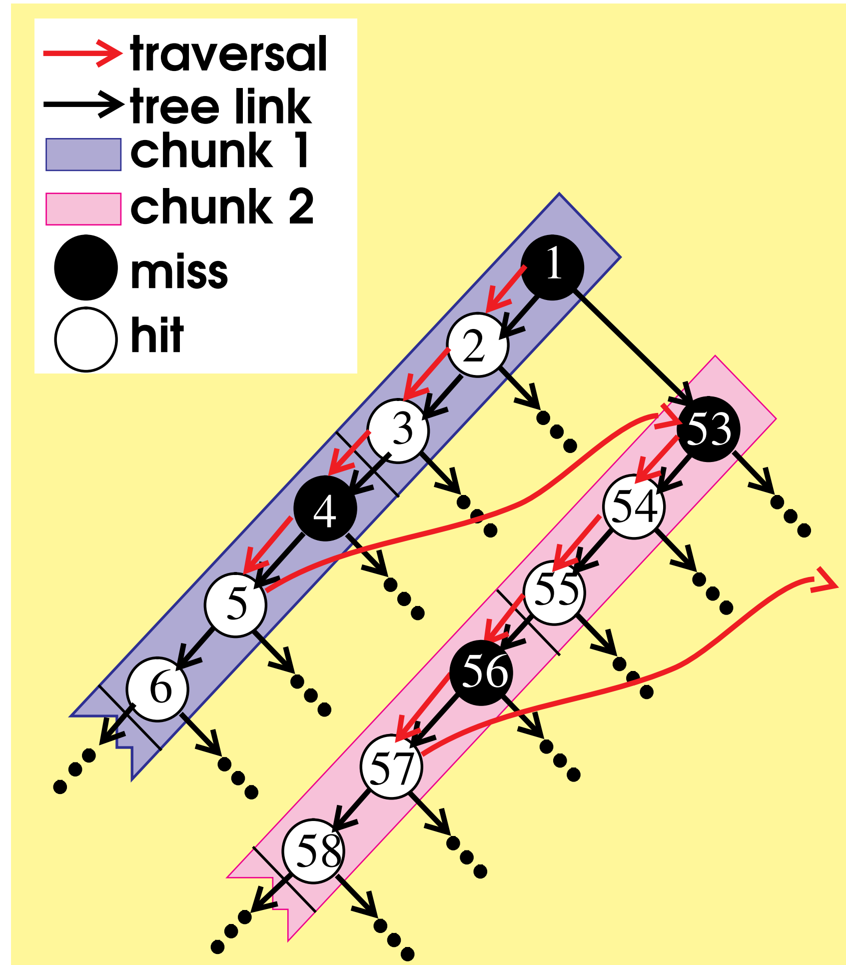
Predict HIT / Actual MISS

Predict MISS / Actual HIT

P = summary C = control-flow

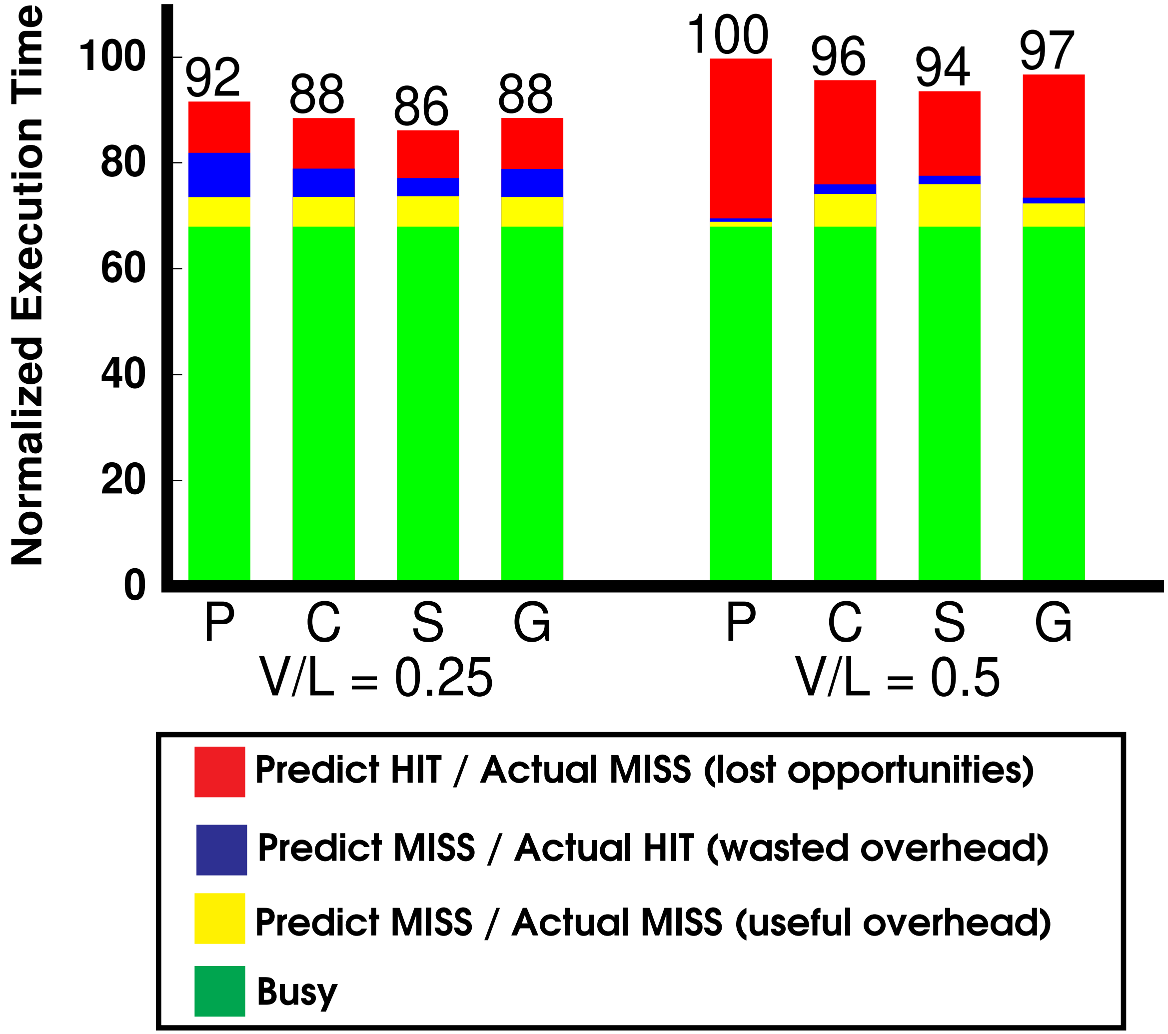
S = self G = global

V / L = overhead to latency ratio



☞ *all three correlation profiling schemes improve prediction accuracy*

Performance of li

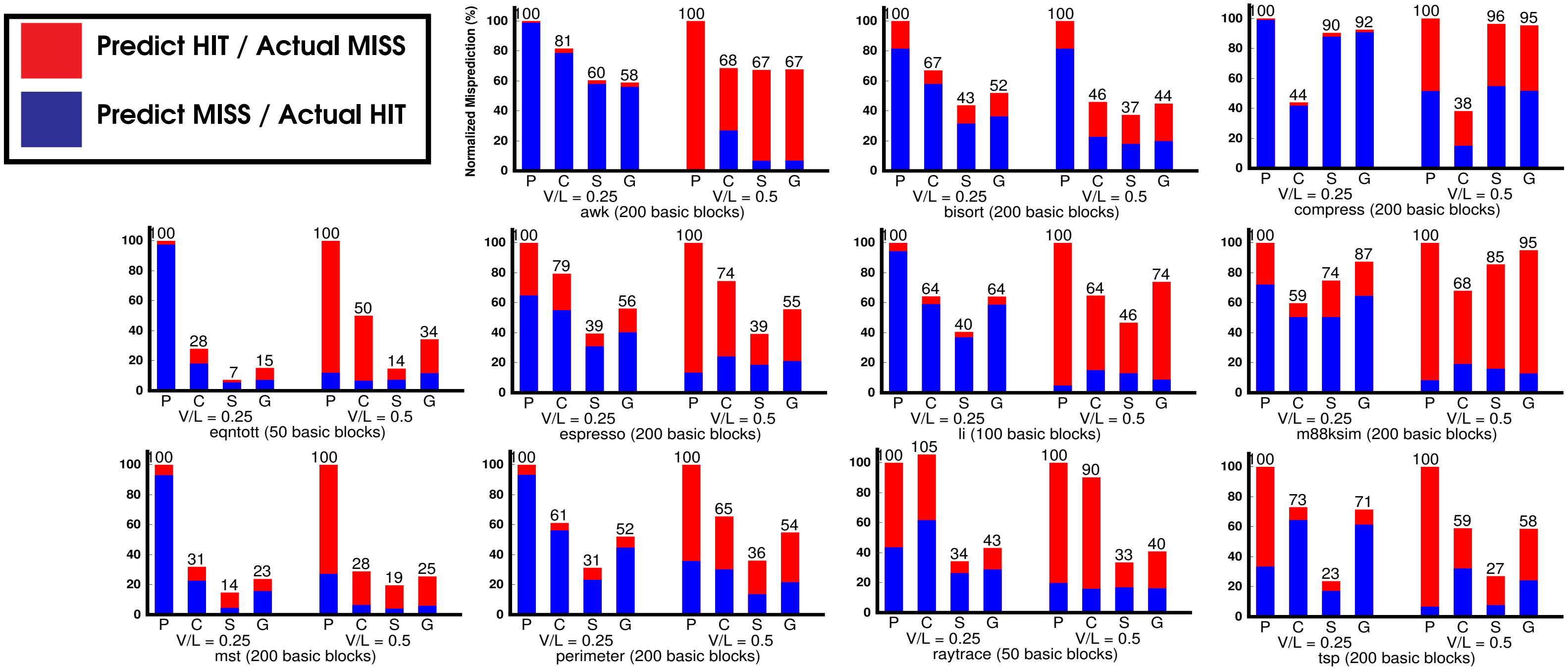


☞ ***Correlation profiling reduces lost opportunities and/or wasted overhead***

Carnegie Mellon

Toronto

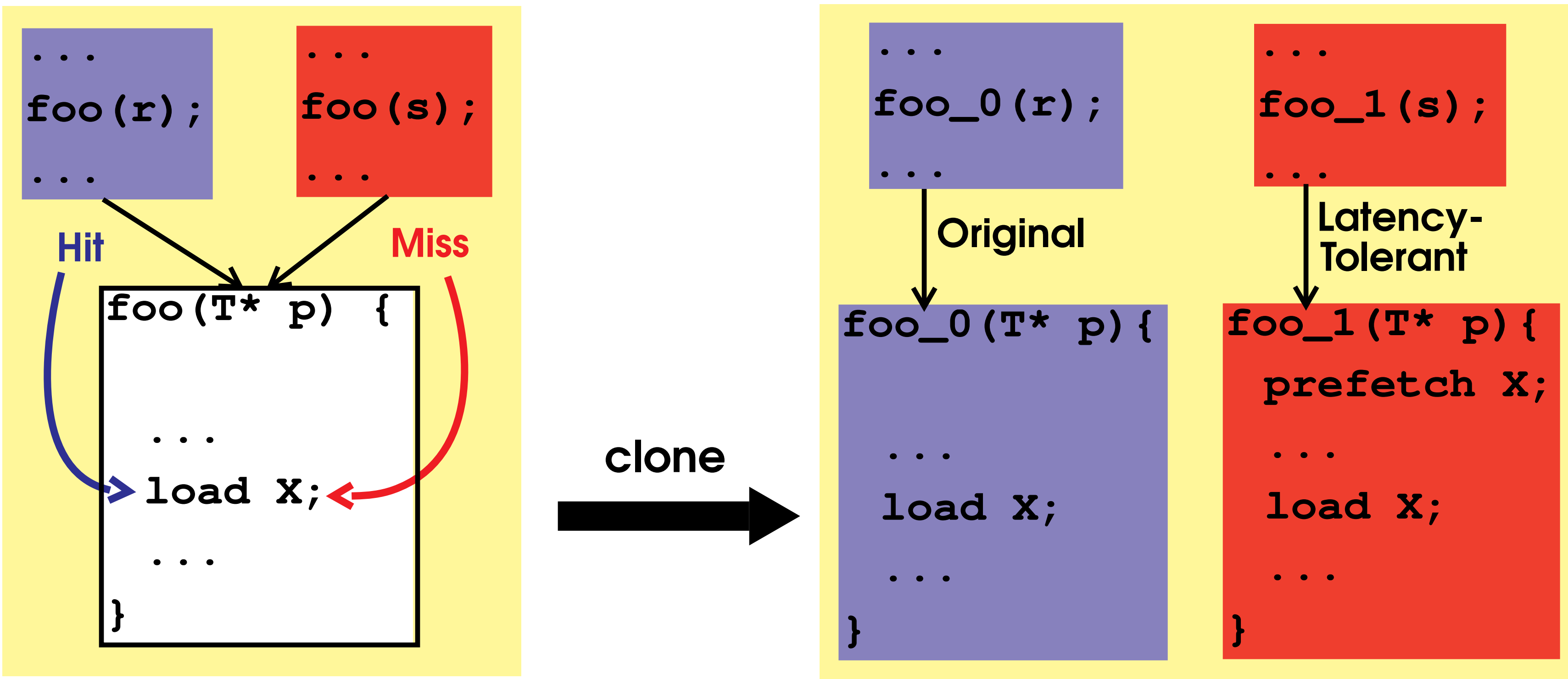
Impact on Prediction Accuracy



- ***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**

Exploiting Correlation Profiling in Practice

- **Easiest case to exploit is control-flow correlation using procedure cloning**



- **Different caching behaviors result from:**
    - different call-site arguments
    - depth in recursion

- **Performance study for prefetching in our technical report**

Conclusions

- **Correlation profiling offers improved accuracy over summary profiling:**
  - **control-flow correlation:** call-chain dependent behaviors
  - **self correlation:** spatial locality, runs of all hits/misses
  - **global correlation:** bursty misses (subsumes self correlation)
- **Correlation profiling can be exploited in practice:**
  - collecting profiling information:
    - informing memory operations (Horowitz *et al*, 1996)
  - isolating dynamic miss instances:
    - procedure cloning (& other code-duplication techniques)

☞ ***correlation profiling may enable more aggressive software-based latency-tolerance techniques***

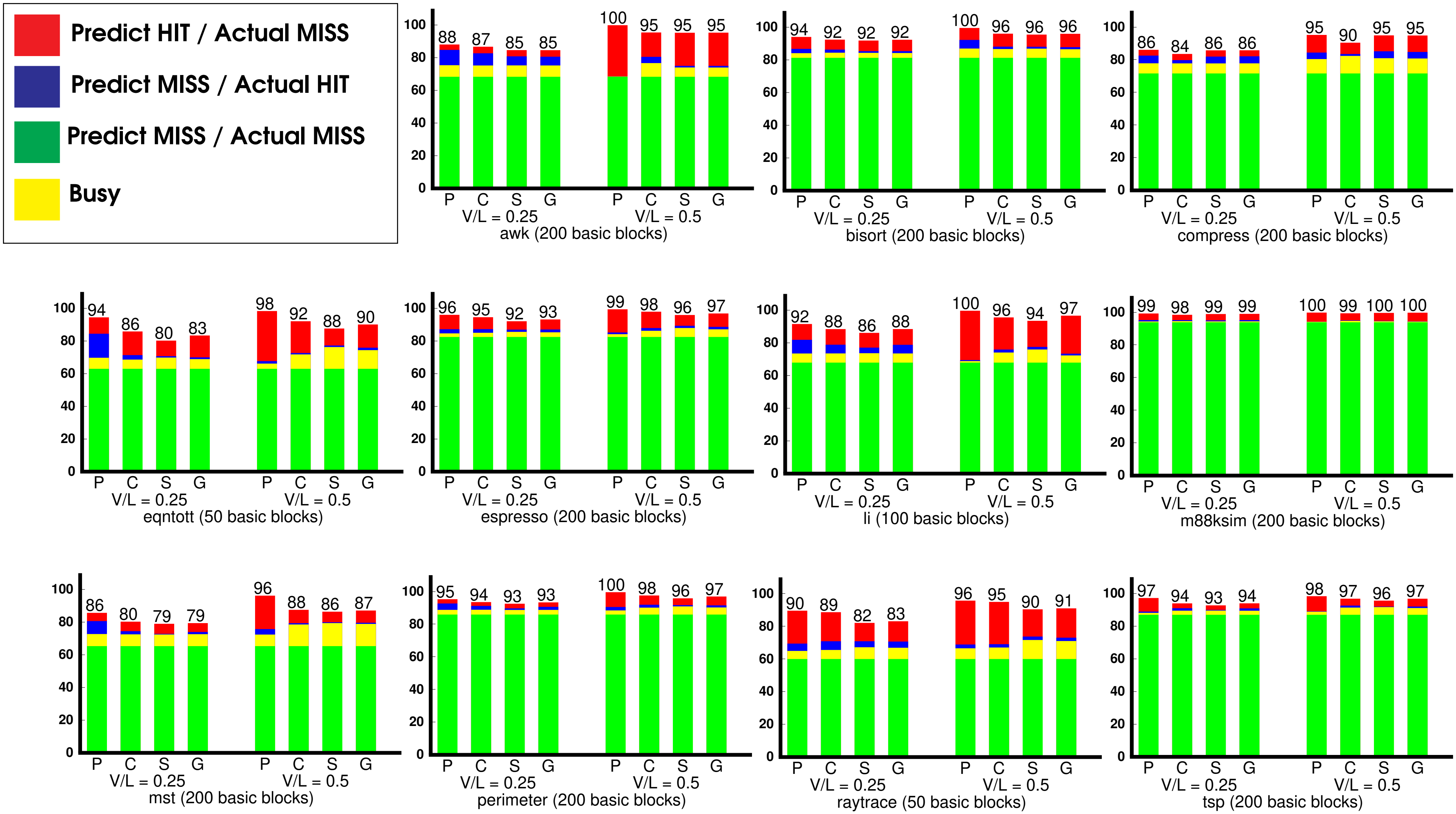
- **Detailed case studies can be found in our technical report:**

CMU-CS-97-175, http://www.cs.cmu.edu/~tcm

Carnegie Mellon

Toronto

Execution Time (L = 50 cycles)



Experimental Methodology

Correlation-profiled loads:

- a load instruction is correlation profiled only if
  - 10% < miss ratio < 90%, *and*
  - among the top 15 loads in terms of total cache miss count

History lengths:

- control-flow correlation: 50 - 200 basic blocks
- self and global correlation: 32 previous cache outcomes

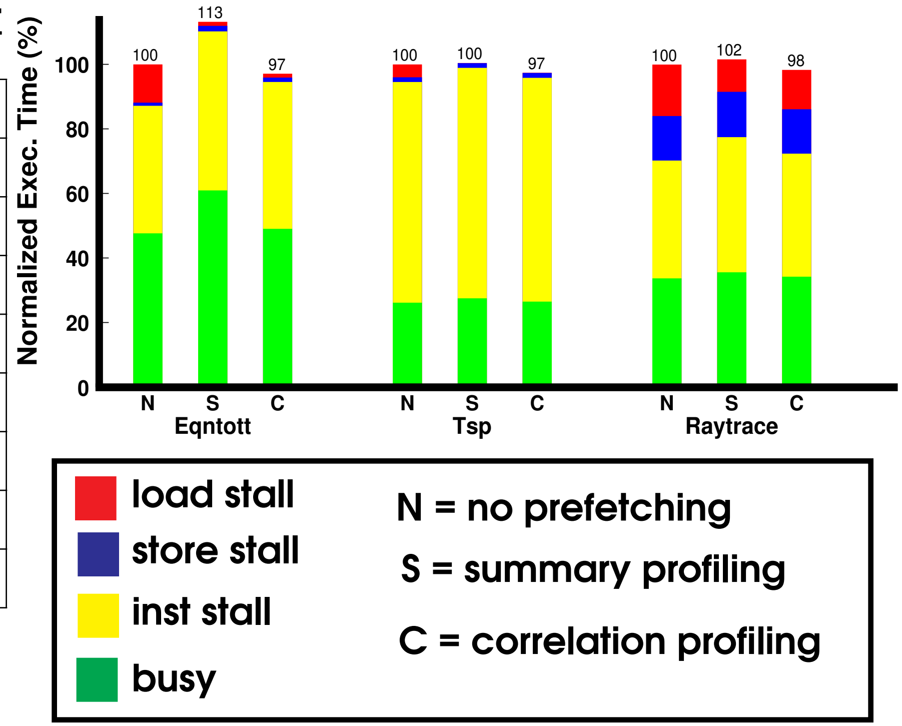
Cache configuration:

- single-level data cache, 2-way set-assoc., 32-byte lines
- cache size ranges from 4 - 128 KB:
  - chose the size resulting in the most correlation-profiled loads

Applying Correlation Profiling to Prefetching

Memory Parameters for our MIPS R10K Simulator

L1 I & D-Caches	32KB, 2-way set assoc.
L2 Cache	2MB, 2-way set assoc.
Line Size	32 B
L1-to-L2 Miss Latency	12 cycles
L1-to-Memory Miss Latency	75 cycles
D-Cache Miss Handlers	8
D-Cache Banks	2
D-Cache Fill Time	4 cycles
Main Memory Bandwidth	1 access / 20 cycles



- **Use procedure cloning to isolate dynamic instances**
- **Adapt the prefetching strategy to the call sites**

☞ ***Correlation profiling reduces overheads without sacrificing the gain***

Dynamic Benchmark Statistics

Suite	Name	Insts	Loads	Load Miss Rate	CP Load Refs	CP Load Misses
SPECint 95	m88ksim	132M	22M (17%)	0.8%	1%	48%
	perl	79M	15M (18%)	12.3%	21%	95%
	go	568M	121M (21%)	7.1%	10%	23%
	ijpeg	1438M	266M (18%)	2.7%	2%	17%
	vortex	2838M	830M (29%)	3.3%	7%	48%
	compress	39M	8M (20%)	3.9%	6%	87%
	gcc	282M	61M (22%)	1.4%	2%	40%
	li	228M	54M (24%)	4.0%	8%	73%
SPECint 92	sc	833M	143M (17%)	9.2%	26%	92%
	espresso	560M	112M (20%)	2.2%	6%	70%
	eqntott	986M	210M (21%)	5.5%	14%	77%
SPLASH-2	raytrace	2105M	588M (28%)	4.8%	10%	53%
	radiosity	996M	236M (24%)	0.4%	1%	32%
Olden	bh	2326M	667M (29%)	1.0%	3%	82%
	mst	90M	14M (16%)	6.9%	17%	91%
	perimeter	123M	17M (14%)	2.3%	5%	88%
	health	8M	2M (25%)	9.0%	20%	84%
	tsp	825M	239M (29%)	1.0%	1%	37%
	bisort	732M	132M (18%)	2.5%	6%	74%
	em3d	420M	73M (17%)	1.4%	4%	98%
	voronoi	263M	87M (16%)	1.3%	4%	57%
Unix Utilities	awk	70M	9M (7%)	7.6%	16%	90%