Cooperative Prefetching:

Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors

Chi-Keung Luk

Department of Computer Science

University of Toronto

Todd C. Mowry

Computer Science Department

Carnegie Mellon University

Instruction Access Latency

- **Is becoming an increasingly important performance bottleneck**

- **Instruction caches are not a complete solution**

- **Accounts for significant stalls in large commercial applications**
  - ☞ >35% execution time wasted on I-cache misses (Barroso *et al* '98)

Experimental Framework

Benchmarks:

- 7 non-numeric applications with large instruction footprints
- include compilers, interpreters, and database programs

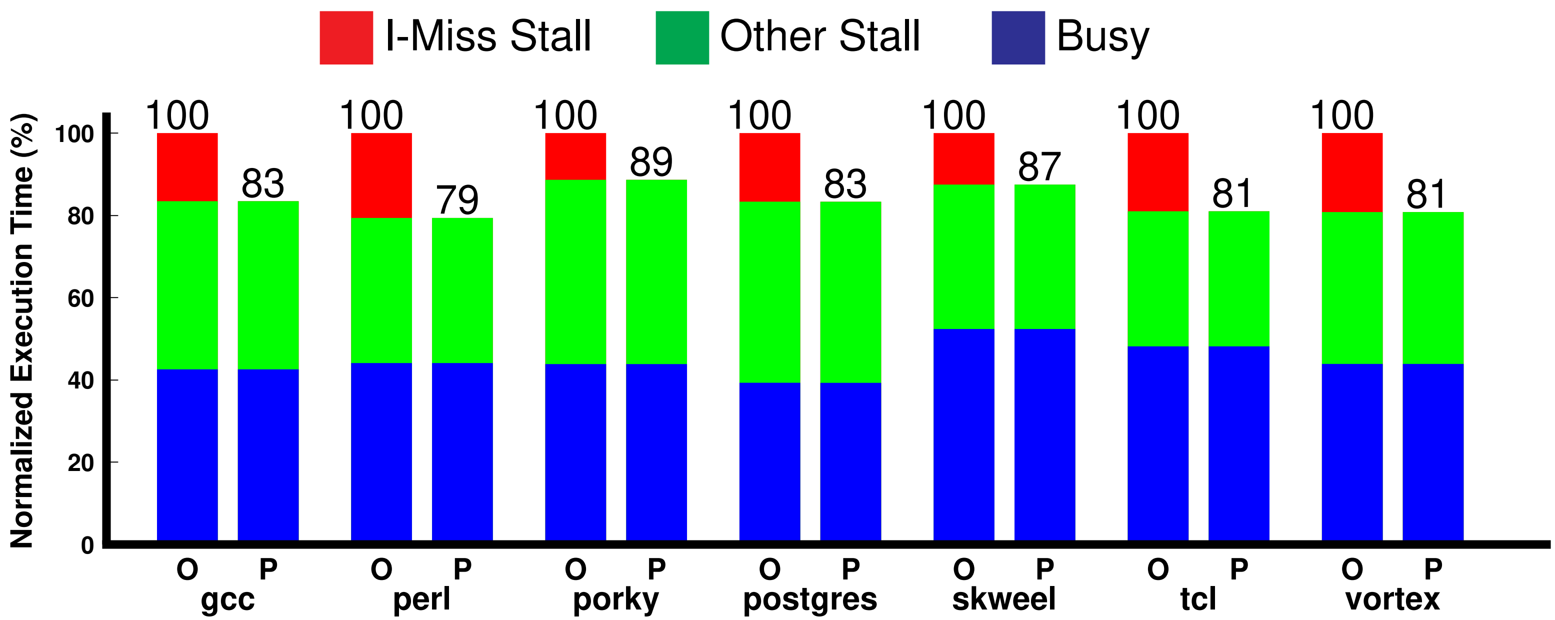
Simulation Model:

- Detailed, cycle-by-cycle simulations of a MIPS R10000-like processor
- Memory hierarchy:
  - 32KB I-cache, 32KB D-cache, 1MB unified Secondary cache

Compiler:

- MIPS CC with -O2 under Irix 5.3

Performance Impact of Instruction Cache Misses



O = Original **P** = Perfect I-cache

- I-cache misses account for 11% - 21% of execution time

☞ one approach: ***instruction prefetching***

State-of-the-Art in Instruction Prefetching

Prefetching sequential accesses:

- Next-*N*-line prefetching (Smith `78, Smith `82)
  - ☞ included in more sophisticated schemes

Prefetching non-sequential accesses:

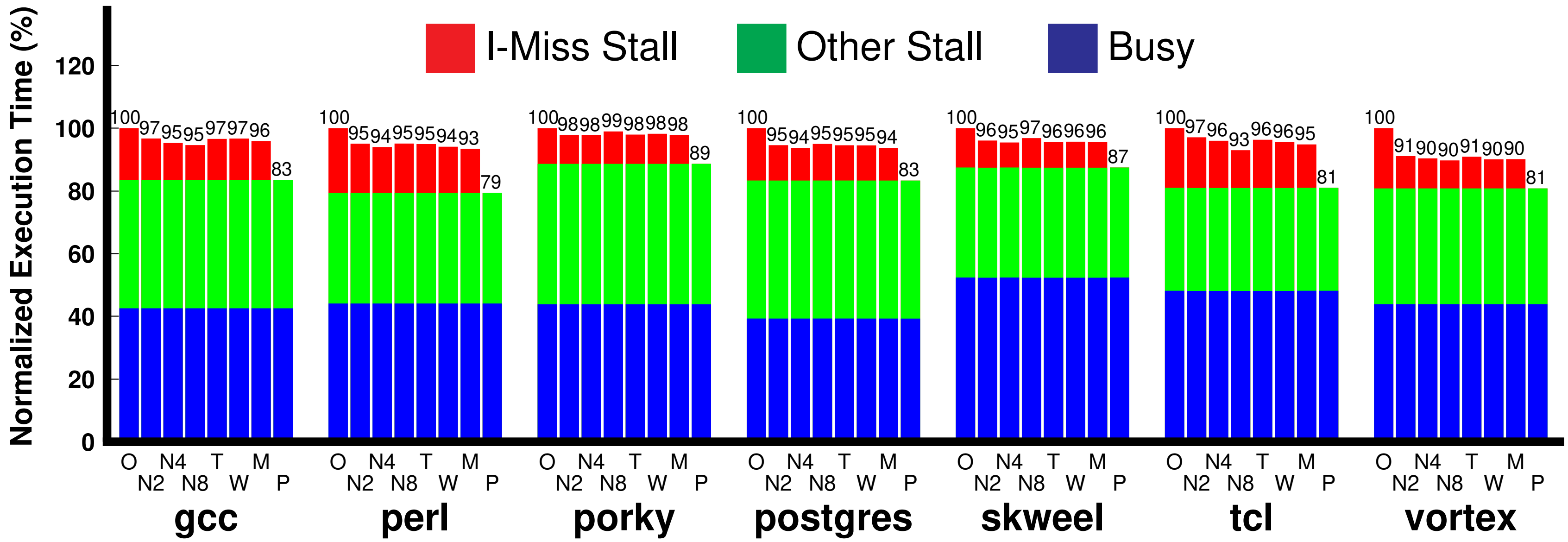
- Target-line prefetching (Smith and Hsu `92)
  - predicts next non-sequential line by current line (history based)
- Wrong-path prefetching (Pierce and Mudge `96)
  - prefetches both paths of conditional branches
- Markov prefetching (Joseph and Grunwald `97)
  - predicts next missing line by current missing line (history based)

Observation:

- These schemes offer little control over the prefetching distance

☞ *difficult to prefetch early enough in modern, fast processors*

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

<u>All schemes offer similarly unsatisfactory performance:</u>

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

<u>Problem:</u> **Prefetches are launched too late!**

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

Overview

- ☞ **Cooperative Instruction Prefetching**
  - **Concepts**
  - **Prefetching Sequential Accesses**
  - **Prefetching Non-Sequential Accesses**
- **Experimental Results**
- **Conclusions**

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:

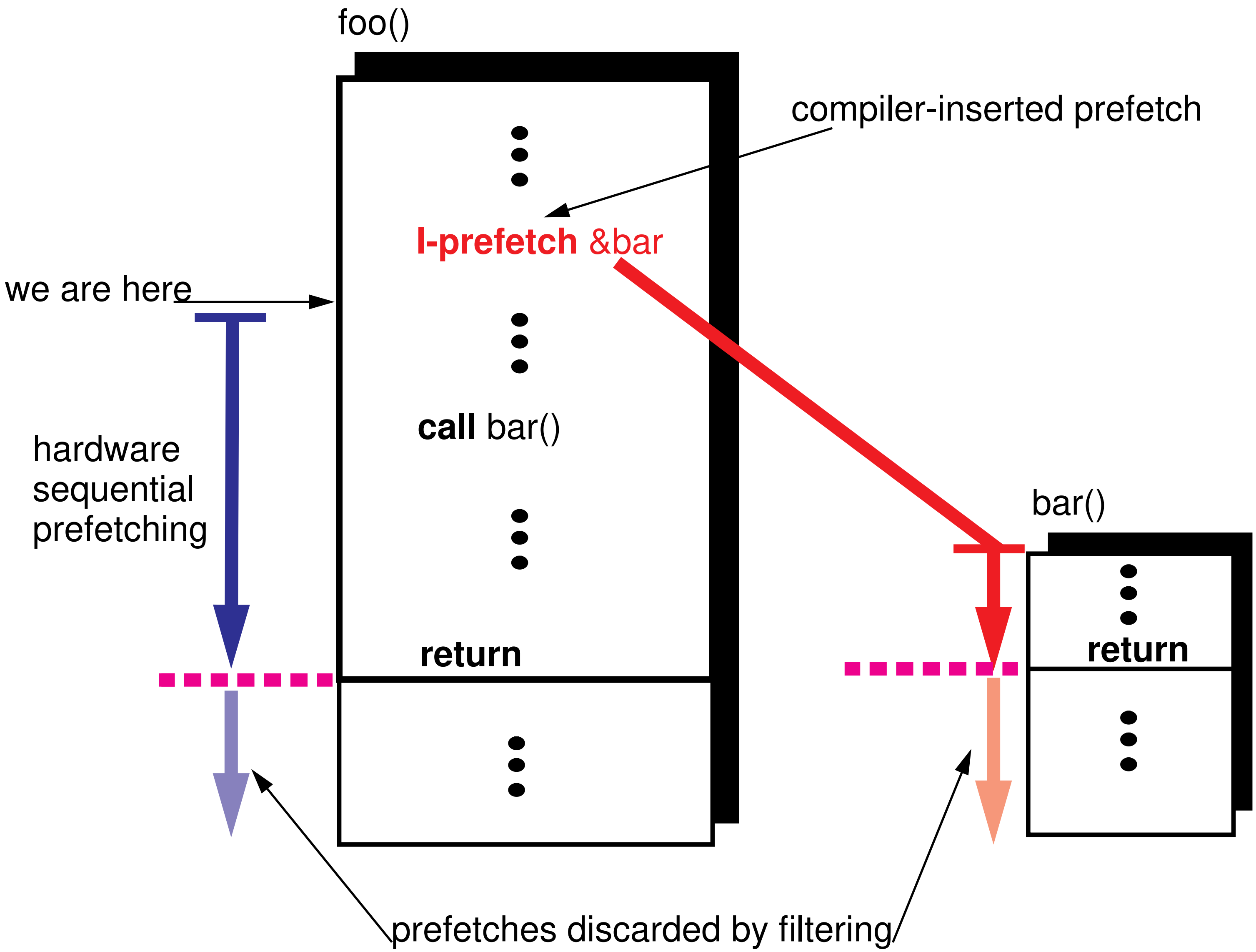
- prefetch filter to eliminate useless prefetches (avoids pollution)
- aggressive sequential prefetching (improves timeliness)

Compiler:

- *automatically* schedules non-sequential prefetches early enough
- minimizes prefetching overhead

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

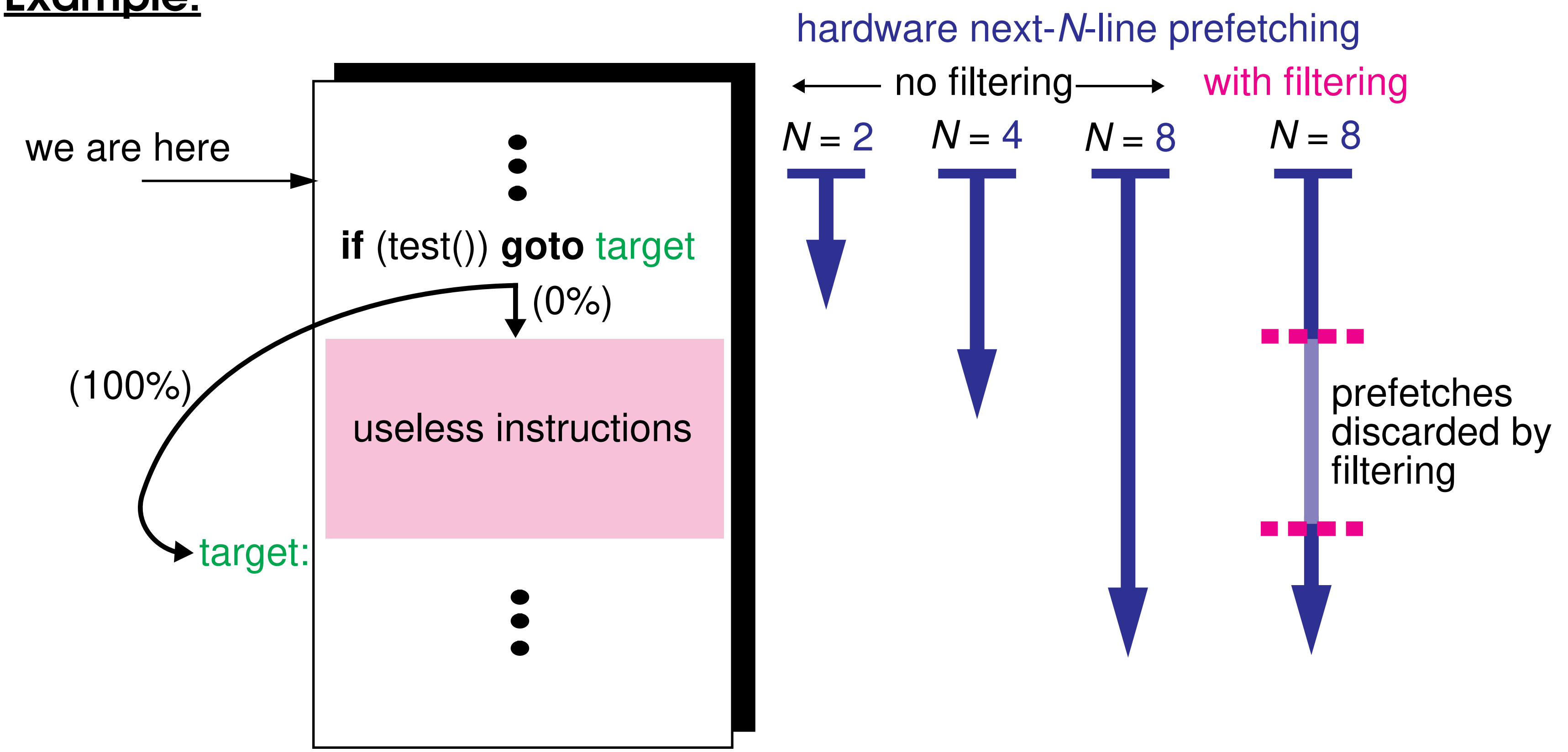
Overall Prefetching Strategies



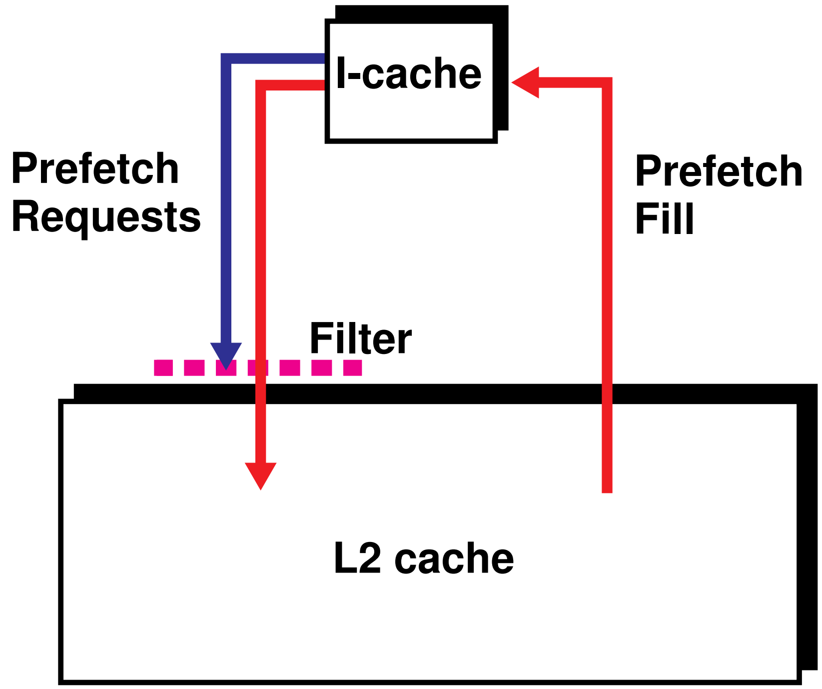
Prefetching Sequential Accesses

- **Hardware issues prefetches for next *N* lines right after entering a new line**
  - *N* is large to increase the prefetching distance (*N* = 8 in this study)
  - prefetch filtering selects a subset of the *N* prefetches to satisfy

Example:



Prefetch Filtering

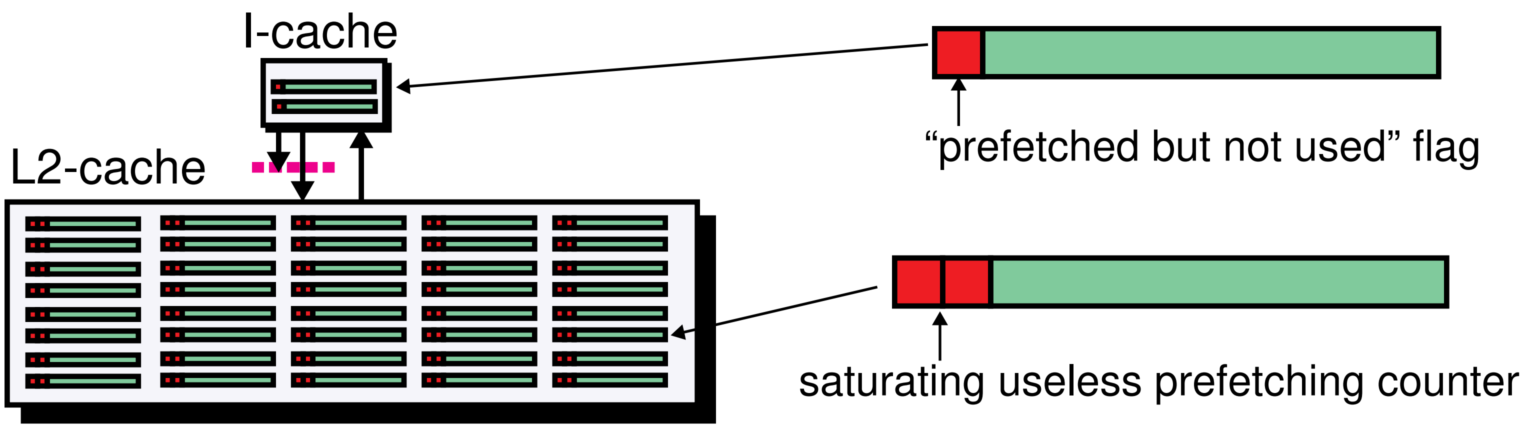


Basic Idea: count useless prefetches of active lines using saturating counters

- counters stored in the tags of the largest cache
    - usually large enough to hold the active working set
- increment count upon replacing unused prefetched line from I-cache
- when count saturates, discard prefetches

☞ applied to both sequential and non-sequential prefetches

Further Details on Prefetch Filtering



Further Details:

- "prefetched but not used" flag in each L1 I-cache line:
  - remembers if the line has been prefetched but not used yet
- saturating counter in each L2 or L3 cache line:
  - counts number of consecutive useless prefetches of the line
- when a prefetched line is displaced from the I-cache:
  - if the "prefetched but not used" flag is set, *increment* the counter
  - if the flag is clear, *reset* the counter
- drop prefetch requests for lines with saturated counters

Prefetching Non-Sequential Accesses

- **Instruction-prefetch instructions are added to the ISA:**

  Format:

  - ordinary: `ipf address`
  - compact: `ipf_c address1, address2`
    - compact format uses smaller offset fields

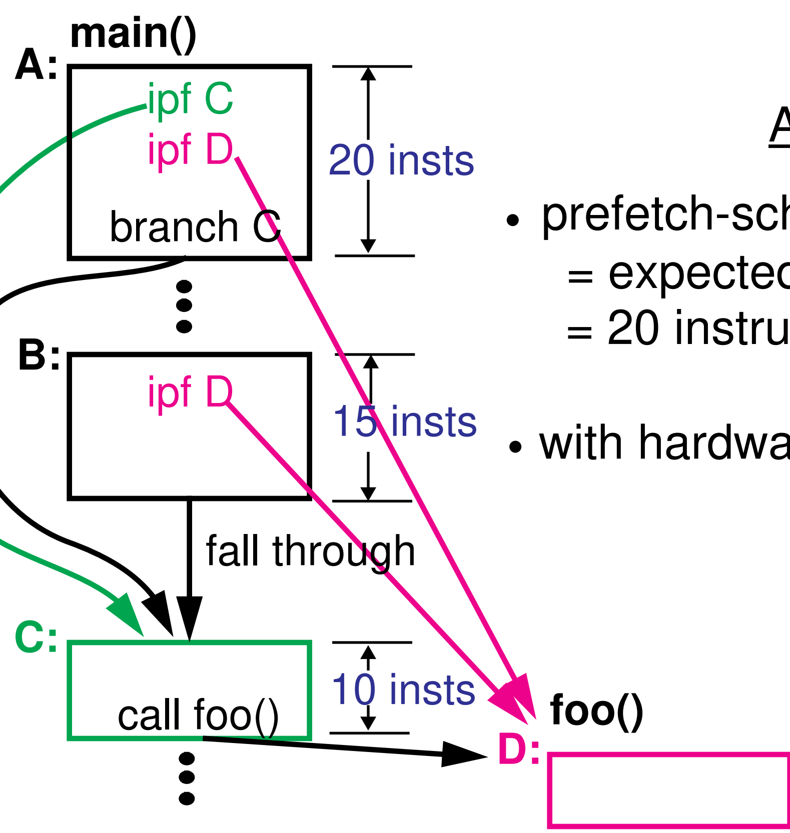
- **Instruction-prefetch instructions complete upon decode:**
  - handled directly by the hardware prefetcher
  - do not consume further pipeline resources (similar to “NOP”)

- **Compiler inserts instruction-prefetch instructions into executables:**
  - automatic compiler algorithm consists of 2 major phases:
    - prefetch scheduling
    - prefetch optimization

Compiler Prefetch Scheduling

- **Examines the control flow across the entire program to insert prefetches**
- **Moves prefetches back by a prefetch-scheduling distance**
    - prefetch-scheduling distance = expected IPC X expected latency

Example:

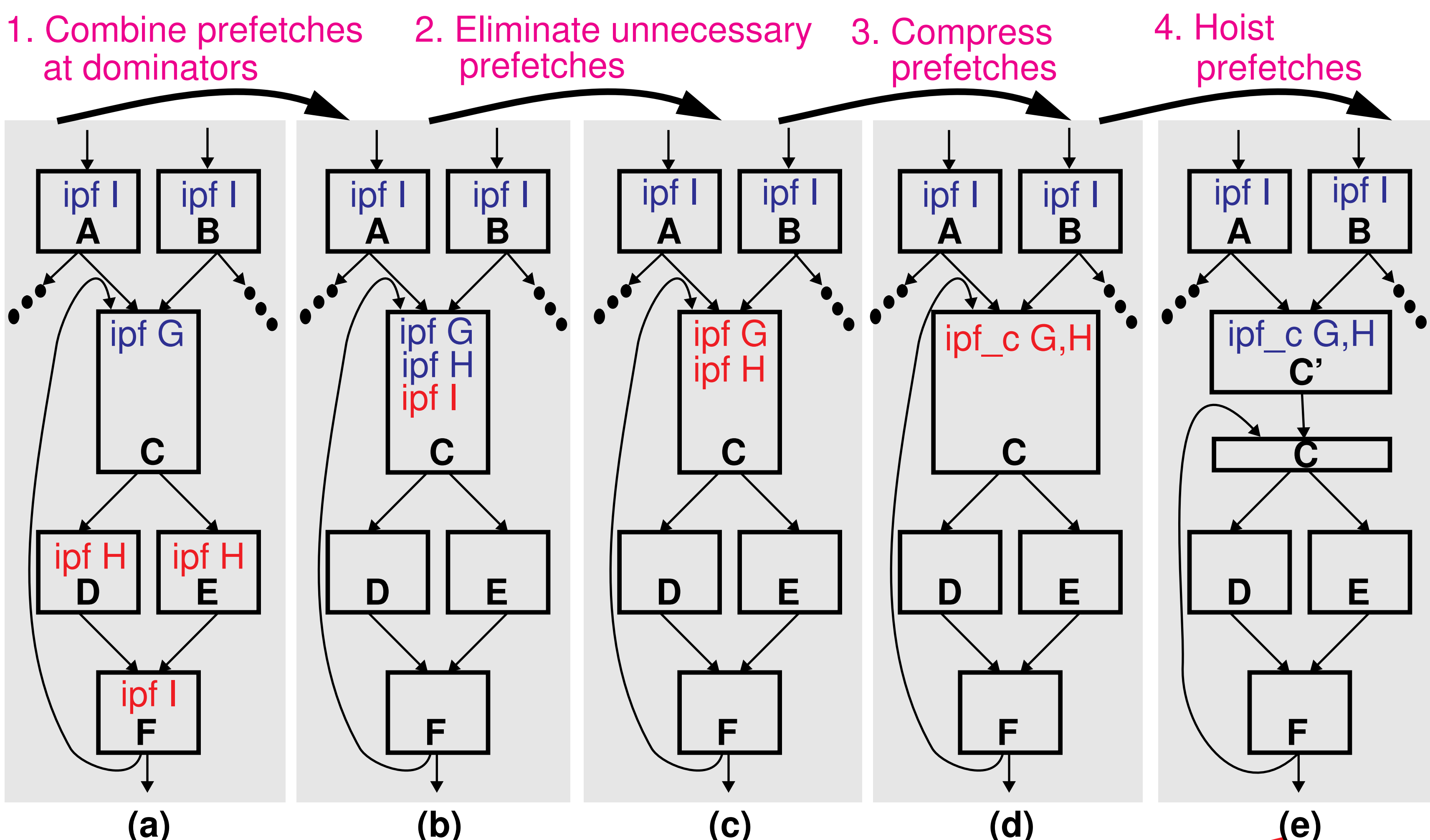


Assumptions:

- prefetch-scheduling distance
  = expected IPC (1.6) X expected latency (12)
  = 20 instructions
- with hardware next-8-line prefetching

Compiler Prefetch Optimization

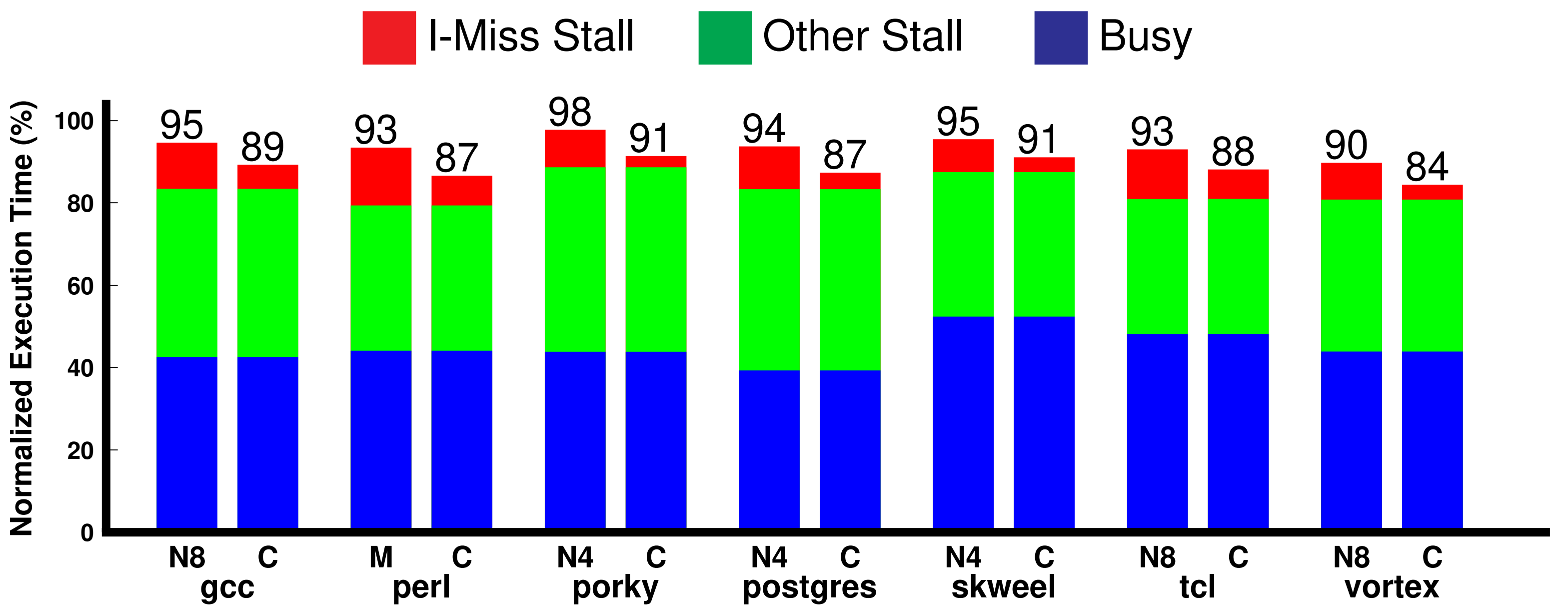
- **Minimizes both static and dynamic instruction overhead**
- **Consists of 4 passes:**



Overview

- ✓ Cooperative Instruction Prefetching
- ☞ Experimental Results
- • Conclusions

Cooperative Prefetching vs. Best Previous Schemes



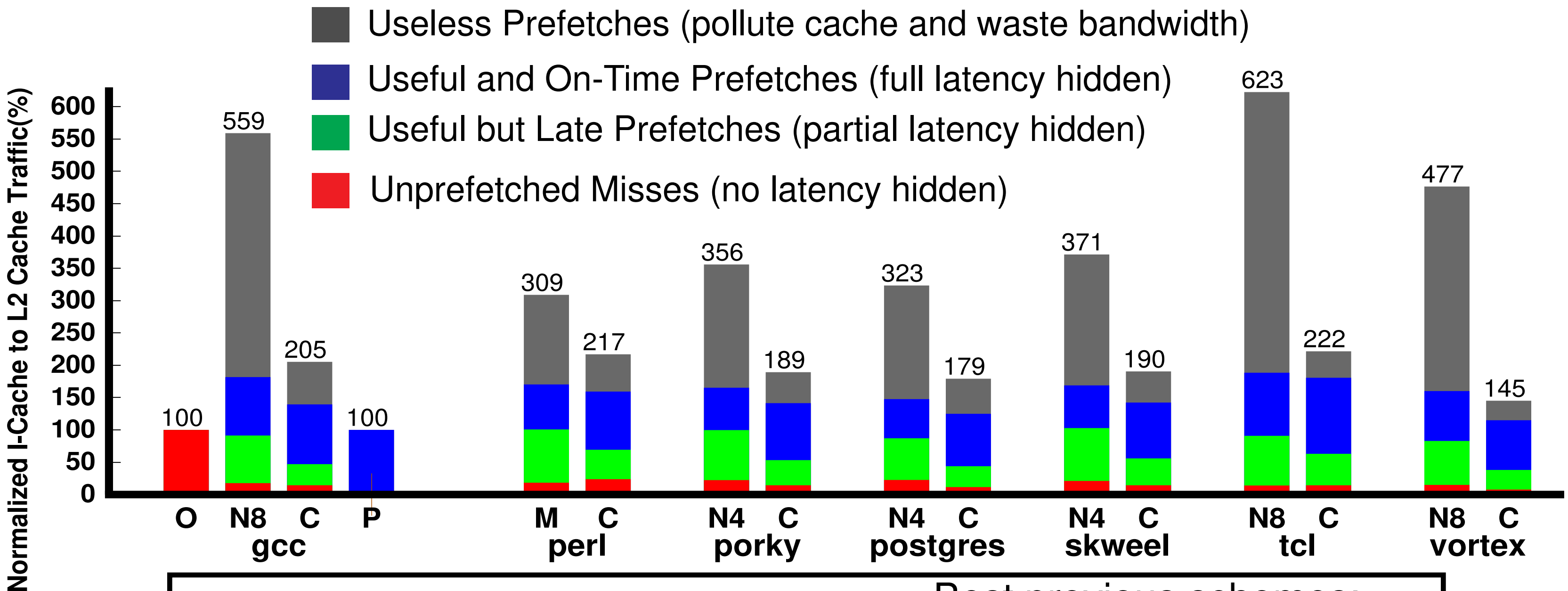
Best previous schemes:

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**

Prefetching Effectiveness



O = Original **P** = Perfect prefetching

C = Cooperative prefetching

Best previous schemes:

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

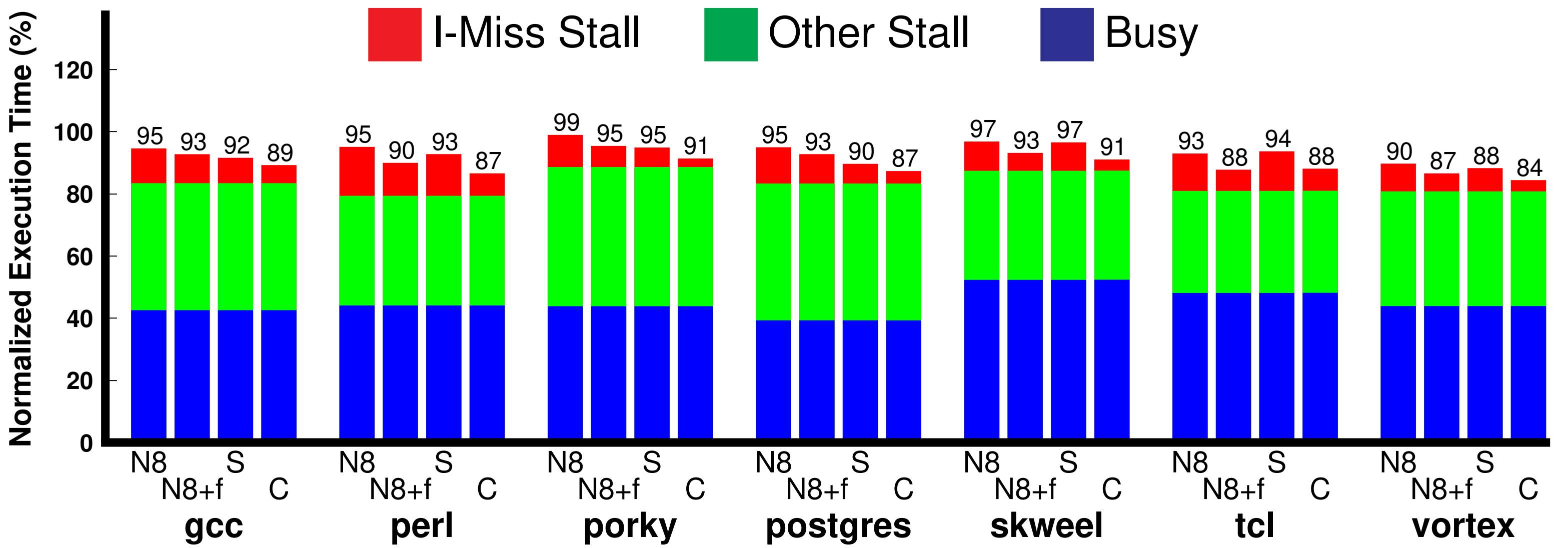
M = Markov prefetching

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**

Importance of Filtering and Software Prefetching



N8 = Next-8-line prefetching without filtering	**N8+f** = **N8** plus prefetch filtering
S = Compiler-inserted prefetching without filtering	**C** = Cooperative prefetching

- Relative importance of prefetch filtering vs. software prefetching varies
- Best performance is obtained by combining both techniques

☞ **both techniques are important and complementary**

Additional Results

- **Compiler optimizations are successful in limiting code expansion**
    - 9% of the original code size on average

- **Cooperative prefetching still performs well with variations in:**
    - L1-L2 miss latency
        - 24% speedup when latency increases to 24 cycles
    - L1-L2 bandwidth
        - 12% speedup when bandwidth decreases to 8B/cycle
        - not bandwidth limited

- **Cooperative prefetching is cost effective**
    - outperforms 64 KB I-cache by 2% on average

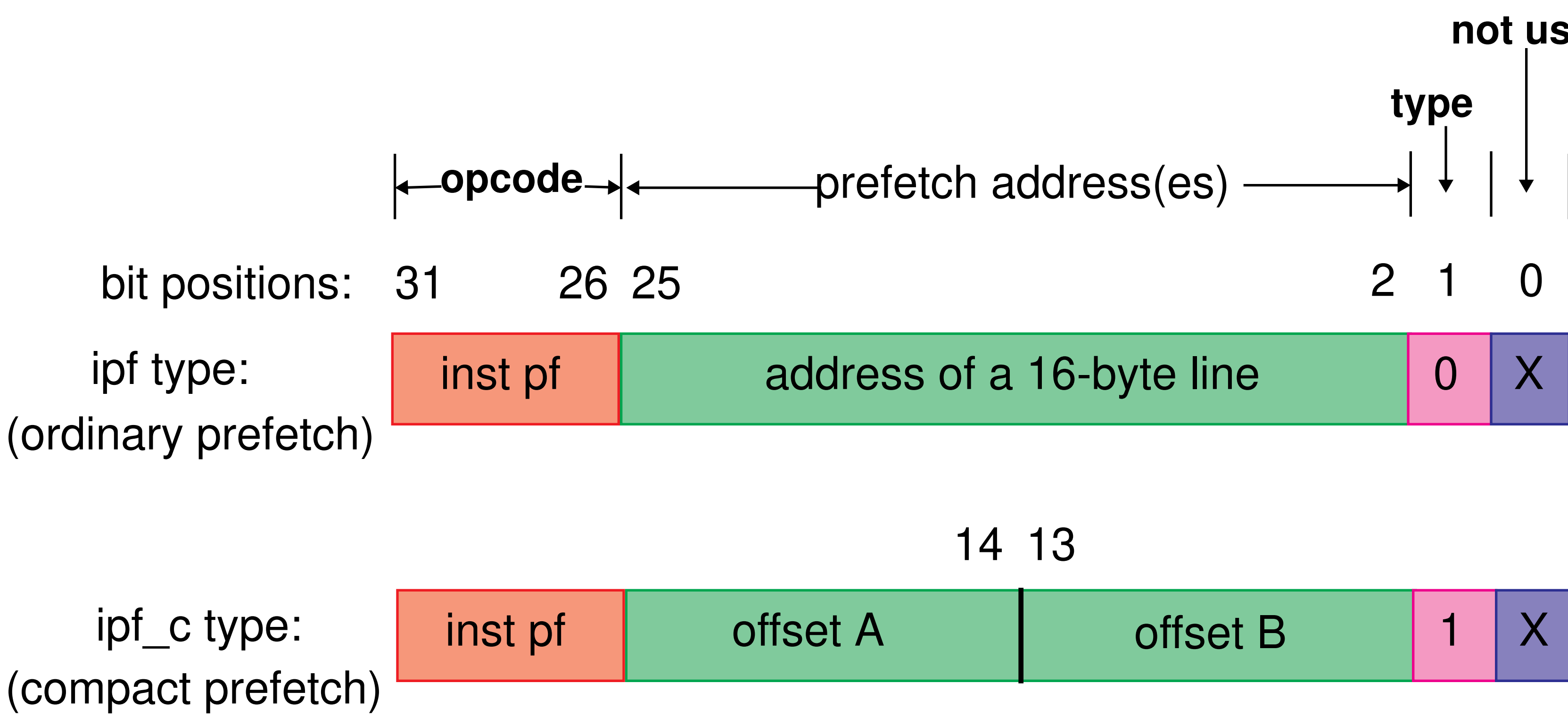
Conclusions

- **Previous schemes launch prefetches *too late***
- **Cooperative prefetching addresses these problems by:**
    - hardware prefetch filtering (avoids pollution)
    - aggressive hardware sequential prefetching
    - compiler-inserted non-sequential prefetching
- **Cooperative prefetching offers significant performance gains:**
    - hides 50% or more of the latency remaining with previous schemes

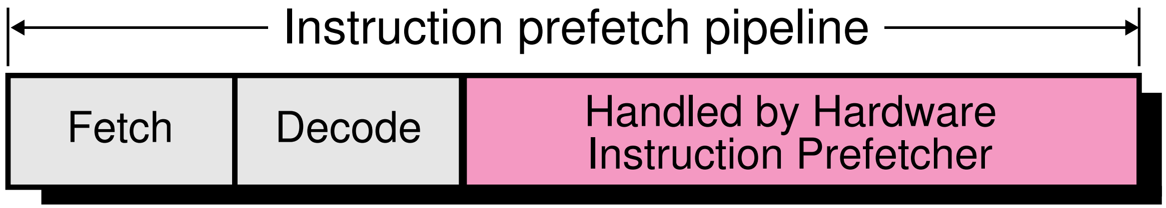
☞ ***Future processors should support instruction-prefetch instructions and the hardware filtering mechanism***

ISA Extensions

- **Based on the jump-type instructions in the MIPS ISA**



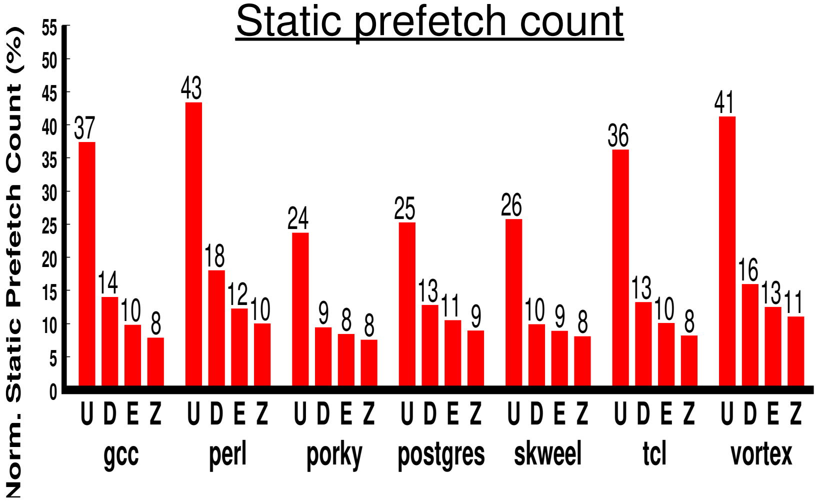
Impact on the Processor Pipeline



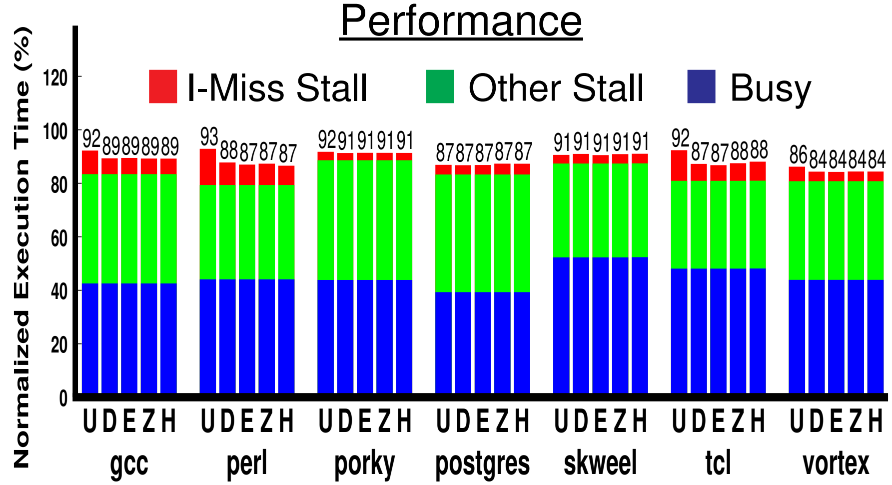
- instruction prefetches are removed once they are decoded
  - ☞ they do not content for most processor resources

Impact of Prefetching Optimizations

Static prefetch count



Performance



U = unoptimized

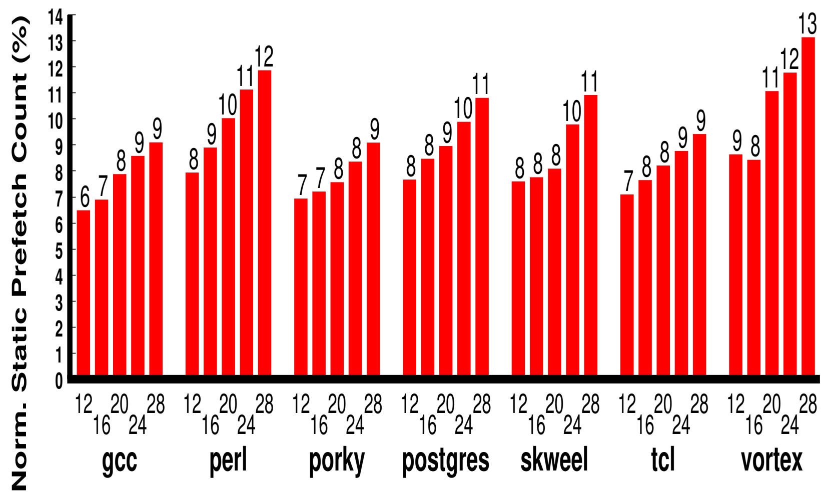
D = combining prefetches at dominators **E** = **D** + eliminating unnecessary prefetches

Z = **E** + compressing prefetches **H** = **Z** + hoisting prefetches

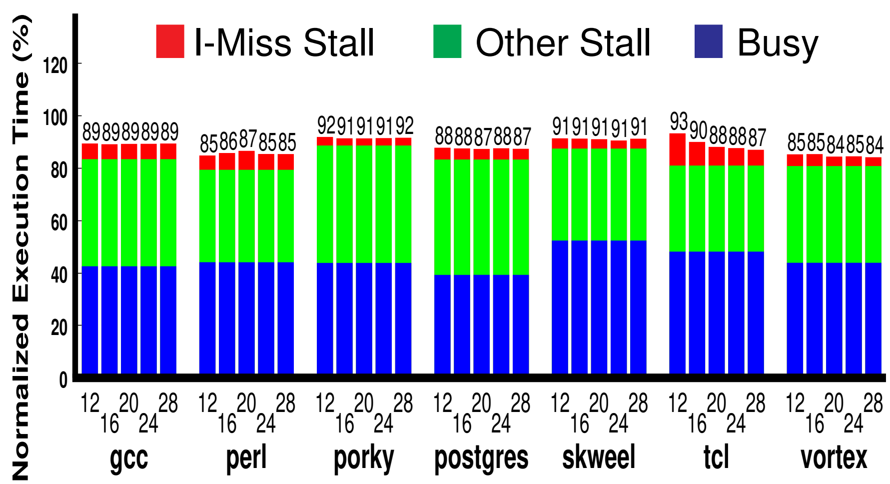
- These optimizations reduce code size significantly
    - on average from 33% to only 9% of the original code size
    - combining prefetches at dominators (**D**) is most effective
- They have less impact on performance
    - optimization **D** still offers noticeable performance improvement

Varying the Prefetch-Scheduling Distance

Static prefetch count



Performance



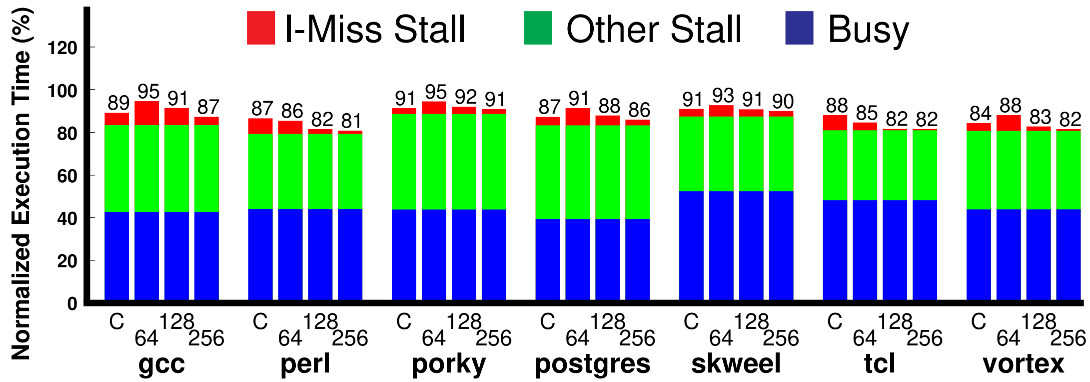
12 = 12 instructions	**20** = 20 instructions	**28** = 28 instructions
16 = 16 instructions	**24** = 24 instructions	

- Increasing the prefetch-scheduling distance increases the code size
  - compiler optimizations limit code expansion to <11% in the **28** case
- Performance is not overly sensitive to the prefetch-scheduling distance

Cost Effectiveness

Cost of cooperative prefetching

Level	Storage
L1	640 B
L2	8 KB



C = 32 KB I-cache with cooperative prefetching
64 = 64 KB I-cache without prefetching
128 = 128 KB I-cache without prefetching
256 = 256 KB I-cache without prefetching

- Cooperative prefetching outperforms 64 KB I-cache by 2% on average
- It also outperforms 128 KB I-cache in 3 applications
- It is within 2% of the performance with 256 KB I-cache in 5 cases

☞ **cooperative prefetching is more cost effective**

Interprocedural vs. Intraprocedural Prefetches

Benchmark	Static Prefetch Count (% of Original Executable Size)	
	Interprocedural	**Intraprocedural**
Gcc	6.3%	1.6%
Perl	8.3%	1.7%
Porky	7.1%	0.4%
Postgres	8.3%	0.6%
Skweel	7.5%	0.6%
Tcl	7.2%	1.0%
Vortex	10.3%	0.7%

☞ **most prefetches are inserted for interprocedural prefetching**