Predicting Instruction Cache Behavior

Frank Mueller (FSU) David Whalley (FSU)
Marion Harmon(FAMU)

Department of Computer Science
Florida State University
Tallahassee, FL 32304-4019

e-mail:
mueller@cs.fsu.edu
whalley@cs.fsu.edu
harmon@vm.cc.famu.edu

Overview

- caches often disabled for real-time due to "unpredictability"
- analysis of instruction cache behavior possible
- *static cache simulation* predicts many references
- allows tighter WET/BET predictions for regular caches
- new architectural feature: *fetch-from-memory* bit
  - speedup factor 3 to 8 over uncached system
  - no loss in predictability

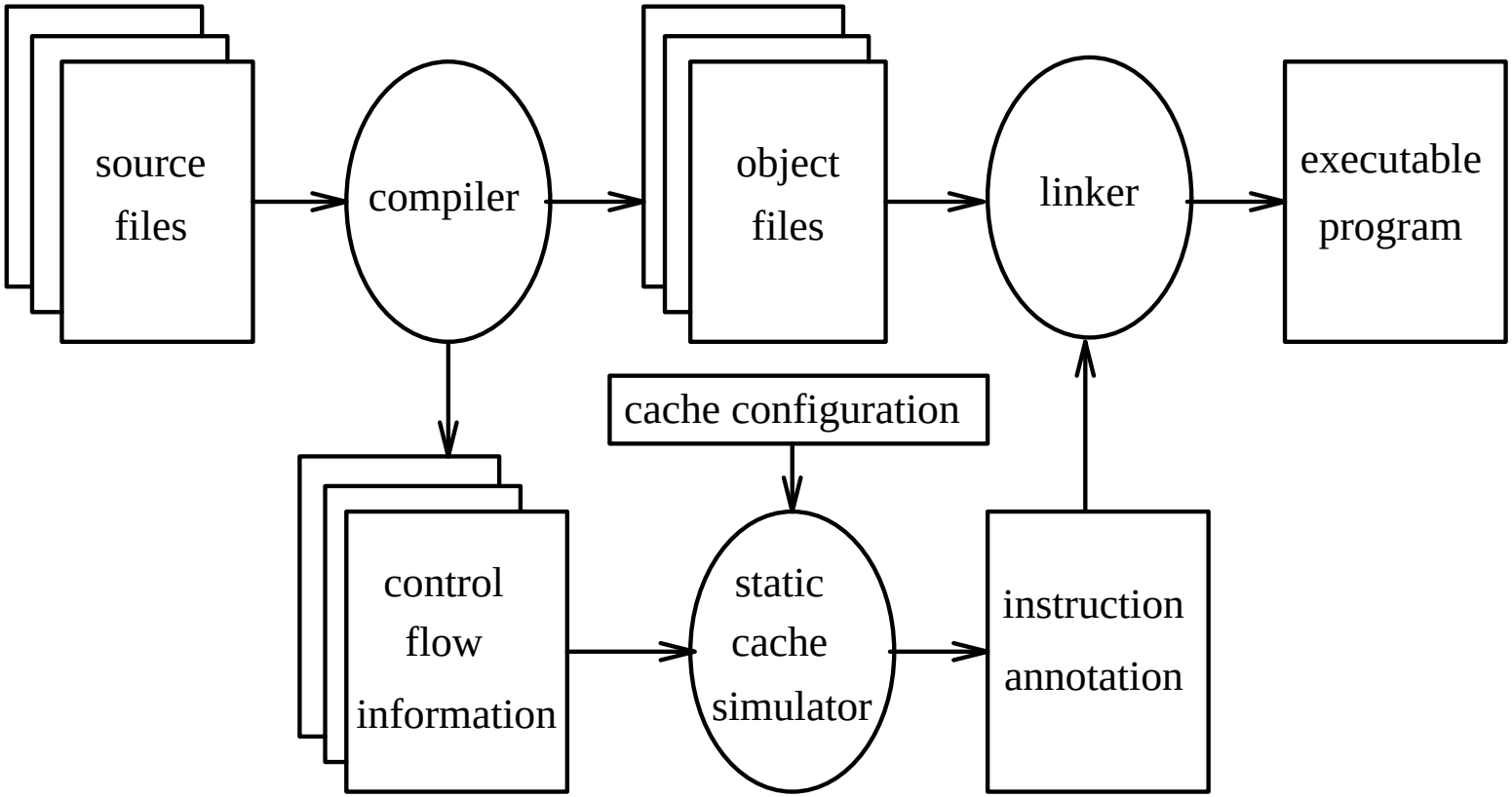
Introduction

- timing predictions required for schedulability analysis
- caches bridge bottleneck between CPU and MM speed
- caches regarded as "unpredictable"
- caches often disabled for hard real-time systems
- CPU speed not fully utilized
- problem will increase in future

Static Cache Simulation

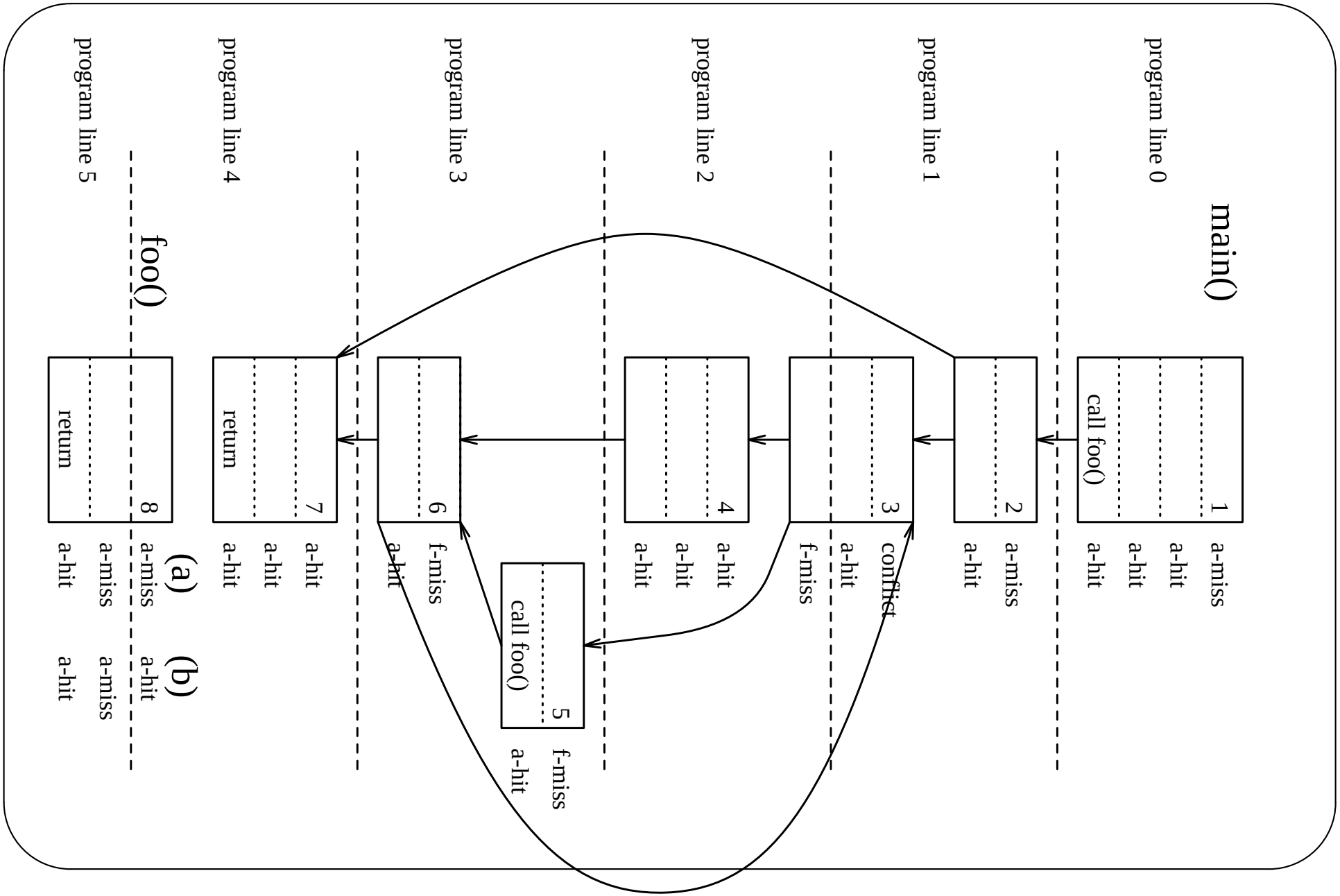
- address of instructions known statically
- predicts large portion of instruction cache references
- uses data-flow analysis of call graph and control flow
- categorizes each instruction
- assumes:
  - direct-mapped caches
  - task: code executed between 2 scheduling points
  - non-preemptive static scheduling
  - currently no recursion allowed

Overview of Static Cache Simulation



Instruction Categorization

- transforms call graph into function-instance graph (FIG)
- performs analysis on FIG and control-flow graph
- uses data-flow analysis algorithms for prediction
- *abstract cache state*: potentially cached program lines
- *reaching state*: reachable program lines
- categories based on these states:
  - always hit
  - always miss
  - first miss: miss on first reference, hit on consecutive ones
  - conflict: either hit or miss (dynamic)


main()
program line 0
program line 1
program line 2
program line 3
program line 4
program line 5
foo()
call foo()
1
2
3
4
5
6
7
8
a-miss
a-hit
a-hit
a-hit
a-miss
a-hit
conflict
f-miss
a-hit
a-hit
a-hit
f-miss
a-hit
f-miss
a-hit
a-hit
a-hit
a-miss
a-hit
return
return
(a)
a-miss
a-miss
a-hit
(b)
a-hit
a-miss
a-hit

- 4 cache lines
- 16 bytes per line (4 instructions)
- instances foo (a) block 8a and (b) block 8b
- 7(1): always hit, spacial locality
- 8b(1): always hit, temporal locality
- 3(3): first miss
- 5(1) and 6(1): group first miss
- 3(1): conflict with 8b(2) conditionally executed

Fetch-From-Memory Bit

- motivation:
  - better performance than uncached systems
  - no loss of predictability
- fetch-from-memory (FFM) bit encoded in instruction
- semantics:
  - FFM set: fetch instruction from MM
  - FFM clear: fetch instruction from cache

Fetch-From-Memory Bit (cont.)

- hardware logic:
  - cache miss: fetch from memory (n cycle delay)
  - cache hit and FFM set: fetch from memory (n cycle delay)
  - cache hit and FFM clear: fetch from cache without delay
- relation to instruction categorization:
  - FFM set iff conflict or always miss
  - FFM clear iff first miss or always hit
- first miss:
  - 1st reference results in cache miss (n cycle delay)
  - consecutive references result in cache hit and FFM clear (no delay)

Measurements

- modified back-end of opt. compiler VPO
- performed static cache simulation
- instrumented programs for instruction cache simulation
- direct-mapped cache simulated
- uniform instruction size of 4 bytes simulated
- cache line size was 4 words (16 bytes)

Static Measurements

Cache Size	cache prediction				
	FFM set	always hit	always miss	first-miss	conflict
1kB	25.19%	71.23%	8.66%	3.69%	16.42%
2kB	21.18%	72.09%	5.88%	7.28%	14.75%
4kB	11.35%	72.40%	4.36%	16.64%	6.60%
8kB	4.73%	72.61%	4.03%	22.77%	0.59%

- cache sizes 1-4kB
- 12 programs with sizes 5-18kB
- FFM set: in DAG call graph and CFG
- others: in FIG
- caches statically predictable for 84-99% of references
- remaining 1-16% due to conflicts

Dynamic Measurements

Cache Size	hit ratio		conflicts	% of exec time	
	bit-enc.	cached	cached	bit-enc.	cached
1kB	71.81%	92.40%	25.38%	39.30%	18.71%
2kB	77.81%	97.49%	21.14%	33.30%	13.62%
4kB	90.73%	99.74%	9.12%	20.38%	11.37%
8kB	98.15%	99.99%	1.76%	12.97%	11.13%

- uncached: simulated disabled instruction cache with 10 overhead for each instruction fetch
- bit-encoded: simulated translation of bit-encoding as discussed
- conventional cached
- 1-19 million instructions executed
- results improve with increasing cache size
- bit-enc.: lower hit ratio than cached (72-98% vs. 92-99%) but much better than uncached!
- bit-enc.: 3-8 times faster than uncached (39-13% of uncached exec time)!
- cached: 5-9 times faster than uncached (18-11% of uncached exec time)
- cached less predictable, bit-enc. as predictable as uncached!
- conflicts source of unpredictability, 25-5%
- results can be still improved if combined with timing tool (4-9 speedup)
- very tight estimating of regular cached system possible with timing tool

Preliminary Timing Results

	Dynamic Worst-Case Measurements		
Name	Observed Cycles	Our Estimated Ratio	Naive Ratio
Matmult	2,917,887	1.00	9.21
Matsum	677,204	1.00	4.63
Matsumcnt	959,064	1.09	4.31
Bubblesort	7,620,684	1.99	8.18

- 8 lines of 16 bytes each, i.e. cache size is 128 bytes
- programs 4-6 times larger than cache
- observed: simulated cached system
- our estimate: timing tool
- naive: uncached system (10 cycles fetch delay per instruction)
- matmult: loops, no if-then-else
- matsum: loops, if-then
- matsumcnt: loops, if-the-else
- bubblesort: inner loop counters depending on outer loop counters
- general problem for timing tools (known number of loop iterations)
- surprisingly tight estimates possible, just as good as without caches

Future Work

- data caching
- recursion
- set-associative caches
- integrate with timing tool to tightly predict WET/BET
- other applications

Related Work

- very little work on predicting cache behavior
- believed to be "unpredictable", too complex to analyze
- timing tools at different code levels:
  - source: Park et. al. (U.W. / Seoul), no caching
  - intermediate: Niehaus et. al. (Amherst), no caching
  - machine: Harmon et. al. (FSU / FAMU), limited caching
- Niehaus: estimated cache hits at abstract level, no method
- architectural modifications by Kirk through cache segmenting
- bit-encoding:
  - McFarling: excl. instr. from cache
  - Chi and Dietz: selected data caching (cache xor register)

Summary

- predicted instruction cache behavior successfully
- designed and implemented static cache simulator to do the job
- regular caches: many references statically known (84-99%)
- bit-encoding: all references predictable, 3-8 times faster than uncached
- tight estimates of WET/BET possible (with timing tool)
- results sufficient for schedulability analysis

INSTRUCTION CACHES CAN FINALLY BE ENABLED
FOR HARD REAL-TIME SYSTEMS