Predicting Negative Cache Interference with Composable Application-Centric Models

Xiaoya Xiang, Bin Bao, Tongxin Bai,
Chen Ding, Trishul Chilimbi

Outline

- Introduction
- Background
- Approximate all-window footprint
- Cache interference prediction
- Evaluation
- Summary

Introduction

- Applications are increasingly run in shared cache
- Asymmetrical effect on performance due to cache sharing
  - equake(<20%) vs vpr(82%)
- Traditional metrics cannot easily explain the asymmetry
- Footprint may help

Background

- What is footprint?
  - Given an execution window in a trace, the footprint is the number of ***distinct elements*** accessed in ***the window***
  - example

k m m n n n

  - compared to reuse distance
    - the number of distinct data elements accessed between this and the previous access to ***the same data***

Background

- What is footprint?
  - Given an execution window in a trace, the footprint is the number of ***distinct elements*** accessed in ***the window***
  - example

    k m m n n n

    window size= 2 footprint=2

  - compared to reuse distance
    - the number of distinct data elements accessed between this and the previous access to ***the same data***

Background

- What is footprint?
  - Given an execution window in a trace, the footprint is the number of ***distinct elements*** accessed in ***the window***
  - example

k m m n n n

window size= 3 footprint=2

  - compared to reuse distance
    - the number of distinct data elements accessed between this and the previous access to ***the same data***

Background

- What is footprint?
  - Given an execution window in a trace, the footprint is the number of ***distinct elements*** accessed in ***the window***
  - example

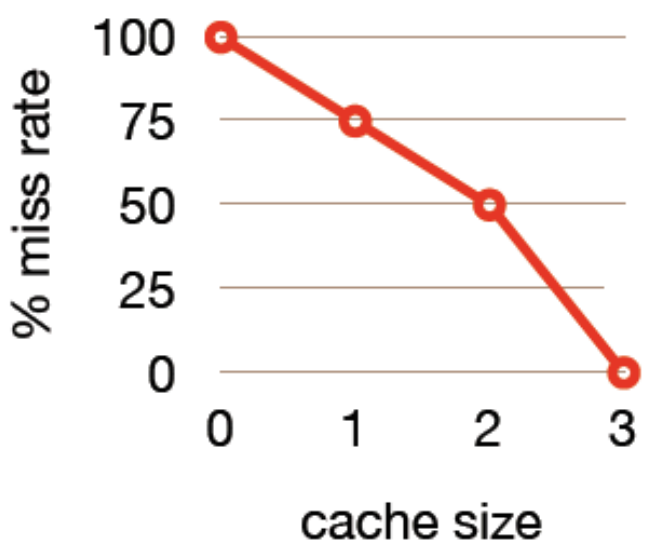
    k m m n n n

    window size= 4 footprint=2

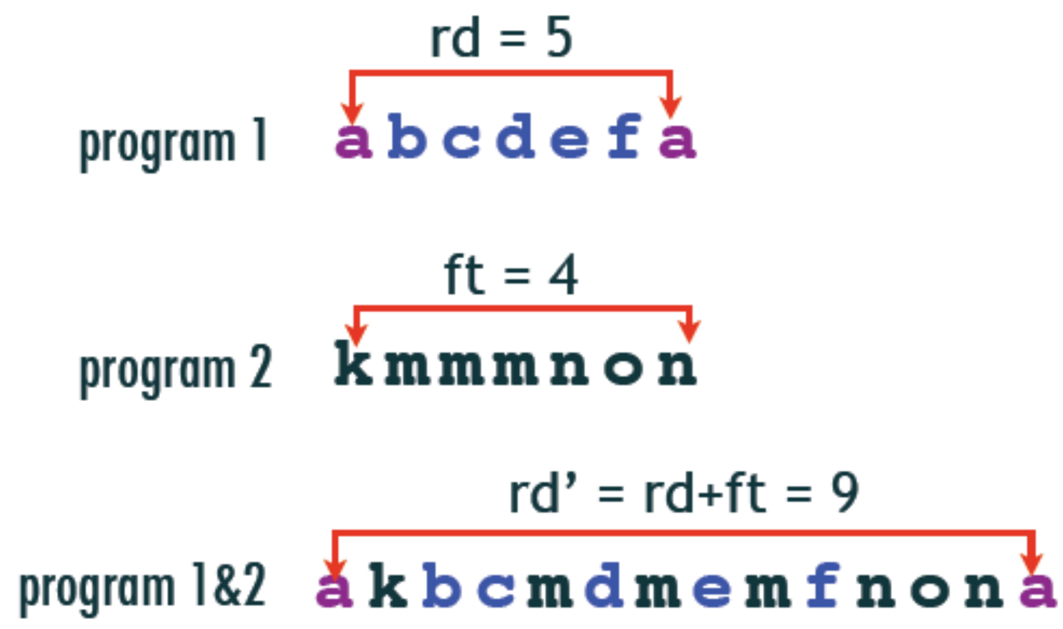
  - compared to reuse distance
    - the number of distinct data elements accessed between this and the previous access to ***the same data***

Locality on shared cache

(a) reuse distances



(b) capacity miss-rate curve computed from reuse distances



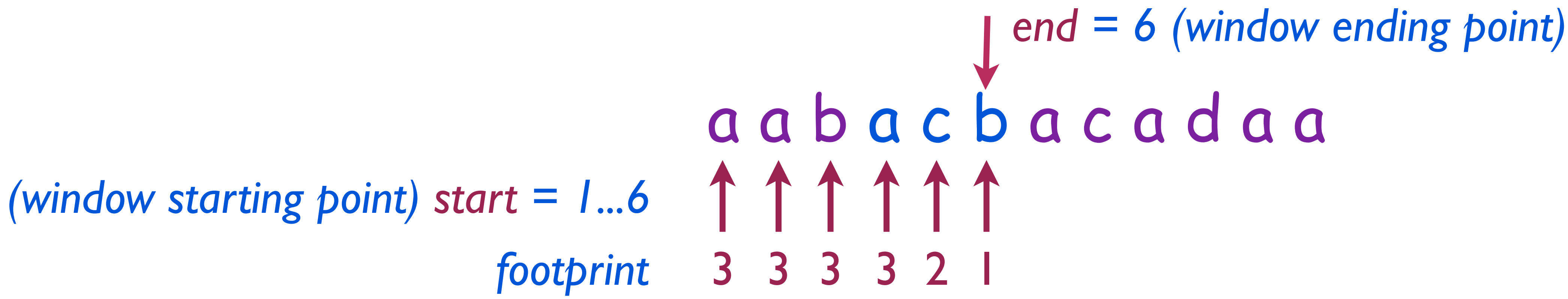
(c) the effect of cache sharing on one reuse distance. The reuse distance of "a" is 5 in program 1. When running concurrently with program 2, it is increased by the footprint of program 2 to 9.

M(A) = P(A's reuse distance >= cache size)

M(A|B) = P(A's reuse distance + B's footprint >= cache size)

All-window footprint

- Given an execution of N run-time data accesses, calculate footprint of all possible windows
- There are N*(N+1)/2 different non-empty windows
- intuitive way
  - traverse the data access trace
  - for each data access, compute the footprint of all windows ending at current access
  - O(N^2)



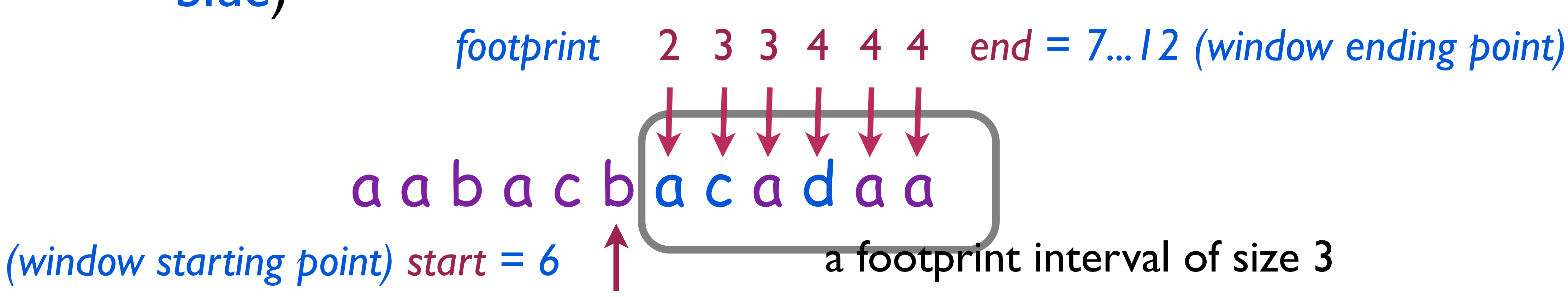
- Observation
    - footprint only changes, when moving left from the endpoint, at the last access of a given element before or up to the window endpoint (in blue)
- NM algorithm: counting footprints instead of counting windows
    - only store the last access of each data (M is the number of distinct data)(Bennett&Kruskal, 1975)
    - fix a footprint, measure the number of windows of that footprint in one step.
    - O(NM)

Further improve by approximation

- NlogM algorithm (Ding and Chilimbi, 2008)
  - Do not care about the exact value of big footprint
    - 1000,000 vs 1000, 001
  - For a relative precision, e.g. 99%, two footprints differ only if their difference is greater than 1% of the smaller one.
  - store only O(logM) data to represent M distinct data
  - O(NlogM)

Further approximation?

- CKlogM algorithm by trace compression (my solution)
  - set a threshold C, e.g. 3. Do not measure footprints smaller than C
  - acceptable since small footprints have little effect on cache sharing
  - divide a trace into a series of intervals called footprint intervals.
  - footprint only changes, when moving right from the startpoint, at the first access of a given element after the window startpoint (in blue)



CKlogM Algorithm

- at most C first accesses of different data within a footprint interval of size C.
- K is the number of footprint intervals in the trace.
- Reduce the asymptotic complexity from O(NlogM) to O(CKlogM)
- Define N/CK as the speedup factor

Speedup for all tests

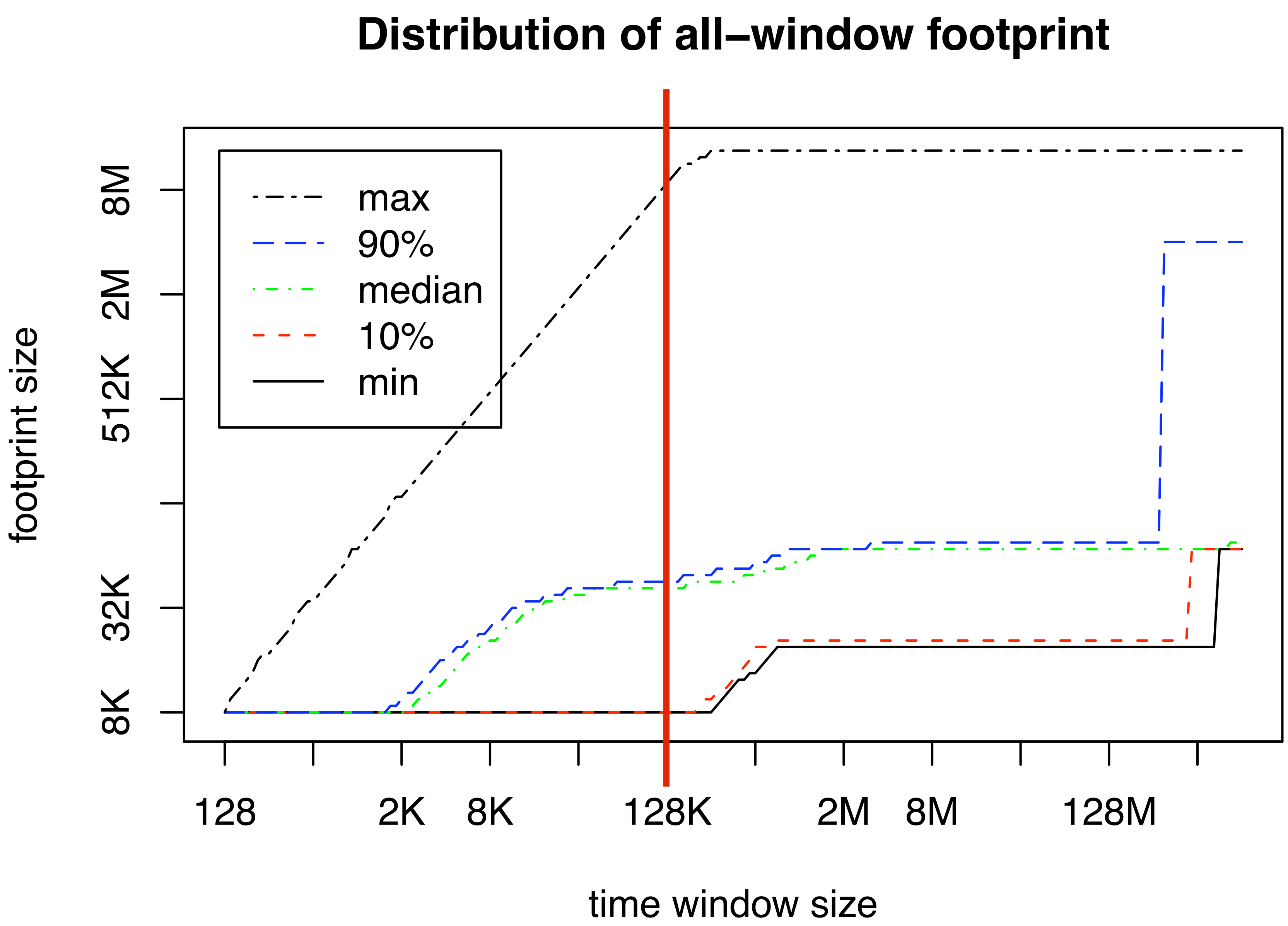
prog.	N	M	NlogM time [sec]	CKlogM C=128 time (speedup)	CKlogM C=256 time (speedup)
gzip	804M	9K	12K	328(35)	246(47)
vpr	298M	5K	4K	84(49)	41(90)
gcc	255M	15K	4K	37(102)	19(198)
mesa	173M	25K	2.6K	10(259)	9(288)
art	1.0B	5K	12K	119(104)	108(114)
mcf	40M	5K	414	52(7.8)	41(10)
equake	342M	40K	5K	42(126)	33(161)
crafty	935M	8K	15K	739(20)	187(78)
ammp	818M	51K	13K	1129(12)	1036(13)
parser	929M	24K	14K	142(101)	98(147)
gap	277M	147K	5K	30(168)	20(252)
vortex	2087M	65K	–	537(N/A)	283(N/A)
bzip2	3029M	60K	–	660(N/A)	565(N/A)
twolf	76M	309	631	3(210)	2(316)
median	320M	12K	5178	47(**101**)	41(**131**) ★
mean	497M	28K	7343	201(**100**)	153(**142**) ★

SPEC2K benchmark statistics

prog.	N (10^9)	M (10^3)	K (10^6)	N/K (10^3)	CKlogM Refs/sec (10^6)
gzip	24	232	7.2	3.4	1.9
vpr	41	159	6.9	5.9	2.9
gcc	16	360	2.3	7.3	3.5
mesa	31	28	1.8	17.5	★ 8.4
art	30	11	12	2.4	1.7
mcf	14	315	27	0.50	★ 0.3
equake	★ 108	167	7.6	14.2	6.5
crafty	41	★ 7.5	25	1.7	1.3
ammp	★ 4.7	51	2.6	1.8	1.1
parser	78	102	21	3.7	1.9
gap	68	★ 787	3.6	19.0	7.3
vortex	31	196	3.0	10.4	4.9
bzip2	42	530	7.8	5.4	2.4
twolf	106	12	48	2.2	1.8
median	36	162	7.4	4.6	**2.1**
mean	45	211	12.5	6.8	**3.3**

C=128
relative precision
=90%

SPEC2K/Gzip(ref input)



1) the y-axle show the value of footprint times cache size (64) since we view each cache line as basic data unit
2) the graph shows statistics of footprints from 10^20 different windows
3) both axles are in log scale

Cache interference prediction

- Shared-cache is a dynamic system
- Circular effect:
  - when two programs A and B are run together, memory access by A affects the performance of B
  - The change in B affects its memory access
  - The change of B's memory access in turn affects the performance of A
- Execution dilation
  - defined as: $$\frac{\text{Execution time of } A \text{ when sharing cache with B}}{\text{Execution time of } A \text{ when running alone}}$$

Construct dilation model step by step

- time model
- dilation definition (i=1, 2)
- cache model in shared-memory system
- combine all to get the iterative model

$$\frac{\delta_1}{\delta_2} = F(\frac{\delta_1}{\delta_2})$$

Construct dilation model step by step

- time model

$$T = T^n n + T^p m^p + T^s m^s$$

- dilation definition (i=1, 2)

- cache model in shared-memory system

- combine all to get the iterative model

$$\frac{\delta_1}{\delta_2} = F(\frac{\delta_1}{\delta_2})$$

Tn	average cost of each instruction
n	# instructions
Tp	average cost of private-cache misses
mp	#private cache misses
Ts	average cost of shared-cache misses
ms	#shared-cache misses

Construct dilation model step by step

- time model

$$T = T^n n + T^p m^p + T^s m^s$$

- dilation definition (i=1, 2)

$$\frac{T^n n_i + T^p m_i^p + T^s m_i^s x_i}{T^n n_i + T^p m_i^p + T^s m_i^s} = \delta_i$$

- cache model in shared-memory system

- combine all to get the iterative model

$$\frac{\delta_1}{\delta_2} = F\left(\frac{\delta_1}{\delta_2}\right)$$

Tn	average cost of each instruction
n	# instructions
Tp	average cost of private-cache misses
mp	#private cache misses
Ts	average cost of shared-cache misses
ms	#shared-cache misses

xi: relative increase in the number of capacity misses in shared cache

Construct dilation model step by step

Tn	average cost of each instruction
n	# instructions
Tp	average cost of private-cache misses
mp	#private cache misses
Ts	average cost of shared-cache misses
ms	#shared-cache misses

xi: relative increase in the number of capacity misses in shared cache

- time model

$$T = T^n n + T^p m^p + T^s m^s$$

- dilation definition (i=1, 2)

$$\frac{T^n n_i + T^p m_i^p + T^s m_i^s x_i}{T^n n_i + T^p m_i^p + T^s m_i^s} = \delta_i$$

- cache model in shared-memory system

$$x_1 = \frac{P\left[d_1 + f_2\left(t(d_1)\frac{cpi_1 \delta_1}{cpi_2 \delta_2}\right) \geq C\right]}{P\left[d_1 \geq C\right]}$$

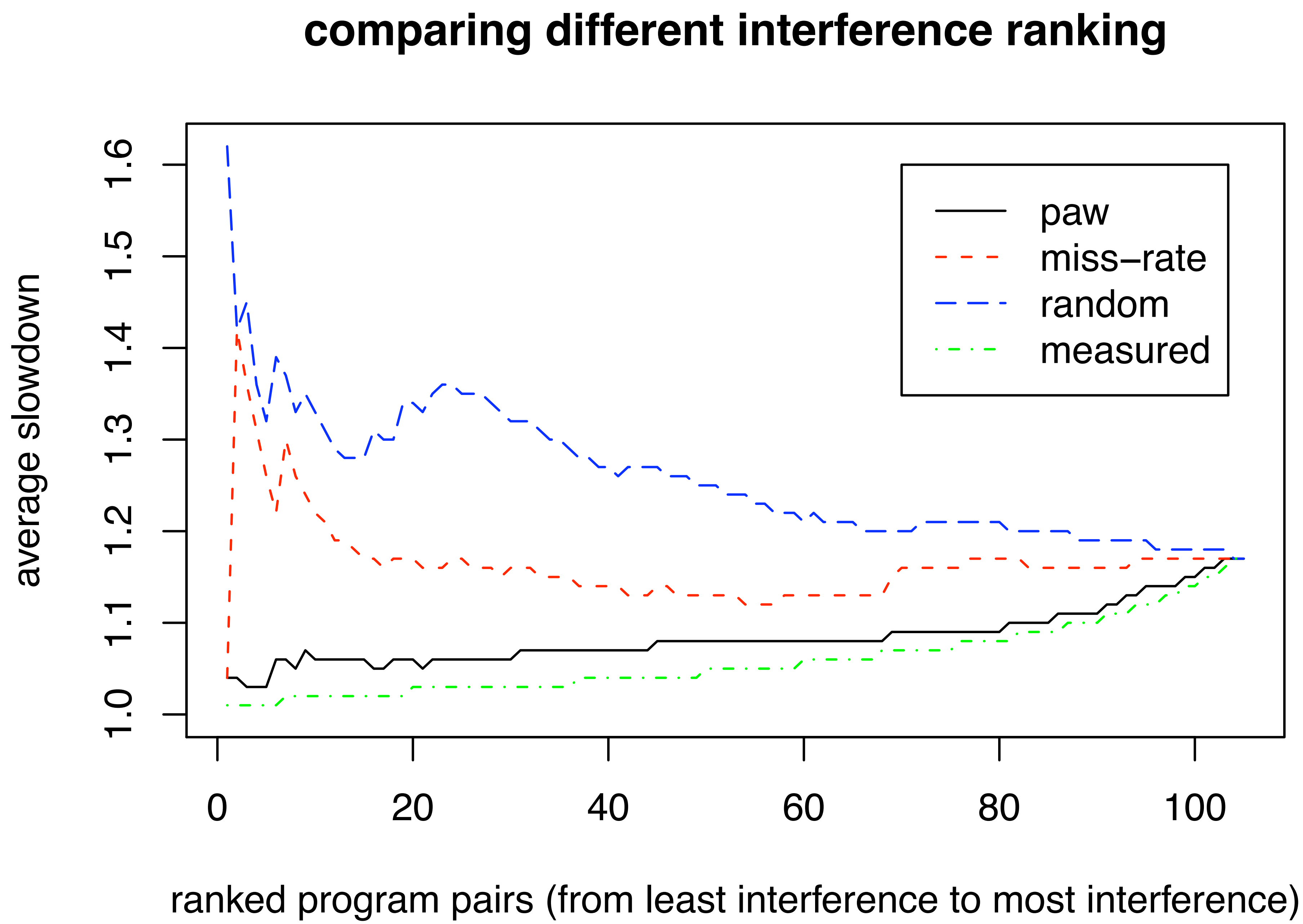
d1: reuse distance of program 1
f2: footprint of program 2
C: cache size
t(d1): a function returning the corresponding window size of reuse distance d1

- combine all to get the iterative model

$$\frac{\delta_1}{\delta_2} = F\left(\frac{\delta_1}{\delta_2}\right)$$

Evaluation

- test set of 15 SPEC2K programs on a dual-core machine (Intel Xeon CPU @ 2.66GHz, 4MB shared cache)
- PAW profile each of the 15 programs in a sequential run to collect reuse distance and footprint information for each program
- Predict dilations of each possible pair(105 in total) and rank it from least performance interference to heaviest
- Alternative ranking methods
  - random ranking: run the standard 15-choose-2 method
  - miss-rate based ranking: based on total miss ratio in sequential run
  - measured ranking: based on results from exhaustive testing of all co-run choices and gives the best possible result.



Y-axis shows the average slowdown for the first x pairs

Summary

- a novel all-window footprint analysis algorithm
  - combines single-window relative-precision approximation and all-window constant-precision approximation to have an asymptotic cost of O(CKlogM).
  - CKlogM algorithm is 100 times faster than NlogM algorithm on average over 14 SPEC2K benchmarks.
- an iterative algorithm to compute the non-linear, asymmetrical effect of cache sharing.
  - a tool for ranking program co-run choices without parallel testing
  - ranking result is close to that from exhaustive parallel testing

- Thanks
- Q&A