ABC-Optimizer: An Affinity-Based Code Layout Optimizer

Chen Ding[1] Rahman Lavaee[1] Pengcheng Li[1]

[1]University of Rochester

November 19, 2013

Background

- Modern software often has a large amount of code.
    - Interpreters, libraries, compilers
- Dynamic execution pattern
    - Especially if it is designed in a modular fashion
- How to optimize code layout in order to exploit instruction locality?

Code Layout Challenges

- Because of the large instruction footprint, instruction misses occur not only for the private L1 icache, but also in the unified cache at lower levels and in TLB.
- Dynamic features such as dynamic typing, meta-programming, and runtime inspection make traditional compiler analysis less effective.

Affinity-Based Solution

- Group elements that are often accessed closed by (have *reference affinity*).
- Two parameters:
    - Footprint distance between accesses (window size)
    - The probability of co-occurrent accesses (co-occurrence confidence)

Reference Affinity: Example

F G F H

- Four footprint windows of size two: $\{F, G\}$, $\{F, G, F\}$, $\{G, F\}$, and $\{F, H\}$.
- Two footprint windows of size three: $\{F, G, F, H\}$ and $\{G, F, H\}$.

Reference Affinity: Co-Occurrence Confidence

F G F H

- Defined for every window size, as:

$$coco(A, B) = \frac{AB.freq}{\max(A.freq, B.freq)}$$

- In this example, for window size two:

$$coco(F, G) = \frac{3}{\max(4, 3)} = 3/4$$

$$coco(F, H) = \frac{1}{\max(4, 1)} = 1/4$$

Affinity-Based Solution

- **Solution:** *Incrementally* group frequently co-occurred elements in relatively small window sizes.
- Leads to a hierarchical partition of elements.
- It's fairly easy to linearize the hierarchical partition.
- Reorder the layout

Applying the Solution to the Problem of Code Layout

- Functions are easy to reorder.
- Trace collection can be done in different levels:
    - Basic block level : unnecessary
    - Function level : insufficient
    - Call level (upon every function entry, and after every call site): appropriate

Single Pass Window Counting

- The new algorithm computes the frequencies, for all window sizes up to a window size limit, in a **single pass**.
- Instead of growing each window at every point, we keep track of a *window list*.
- The window list is a two-level doubly linked list.
  - Each upper level element is a *partial* window.
  - Each lower level element is a function record.

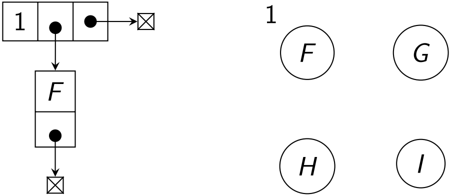
Execution of the Algorithm on an Example Trace

F	G	F	G	F	H	I
x			x			

Window Creation at Sampling Point

F G F G F H I

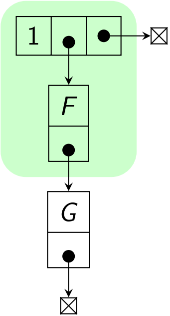
x x

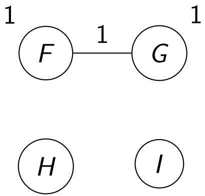


Window Growth

F G F G F H I

x x

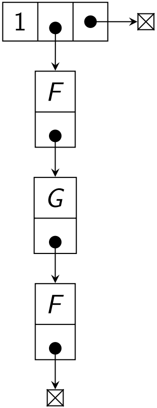


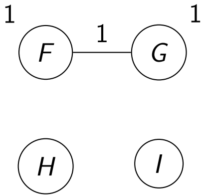


Attempt for Window Growth

F G F G F H I

x x

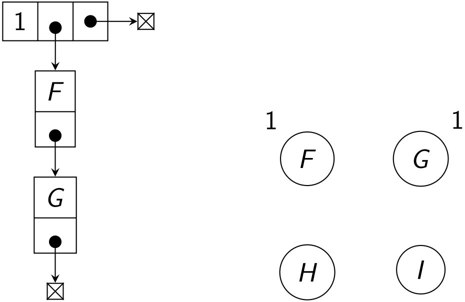




No Window Growth (Cleanup)

F G F G F H I

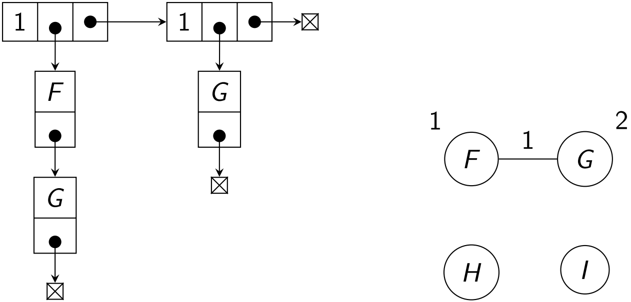
x x



New Window creation at Sampling Point

F G F G F H I

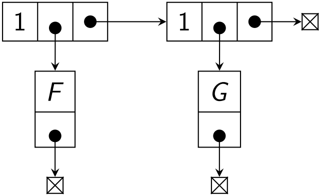
x x

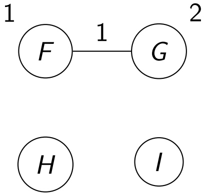


No Window Growth (Cleanup)

F G F G F H I

x x

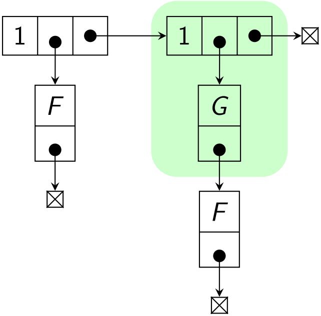


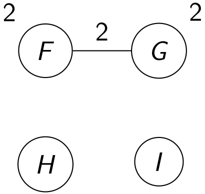


Window Growth

F G F G F H I

x x

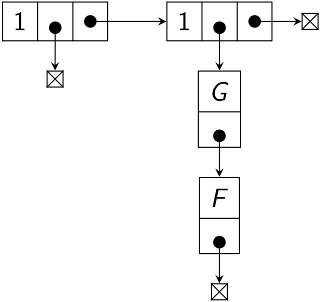


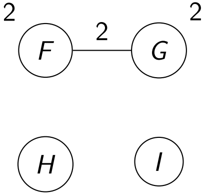


Cleanup Record

F G F G F H I

x x

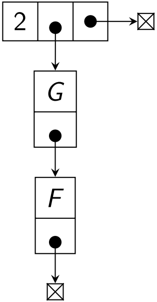


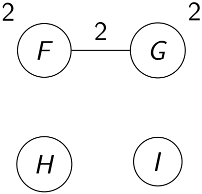


Cleanup Window and Add Window Counts

F G F G F H I

x x

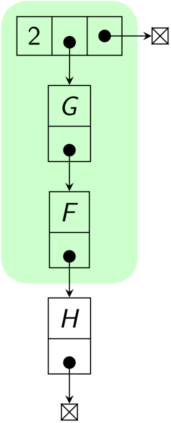




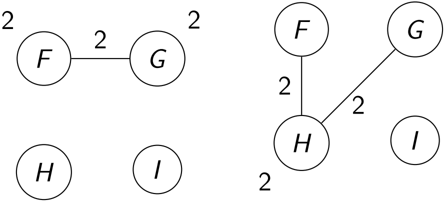
Window Growth

F G F G F H I

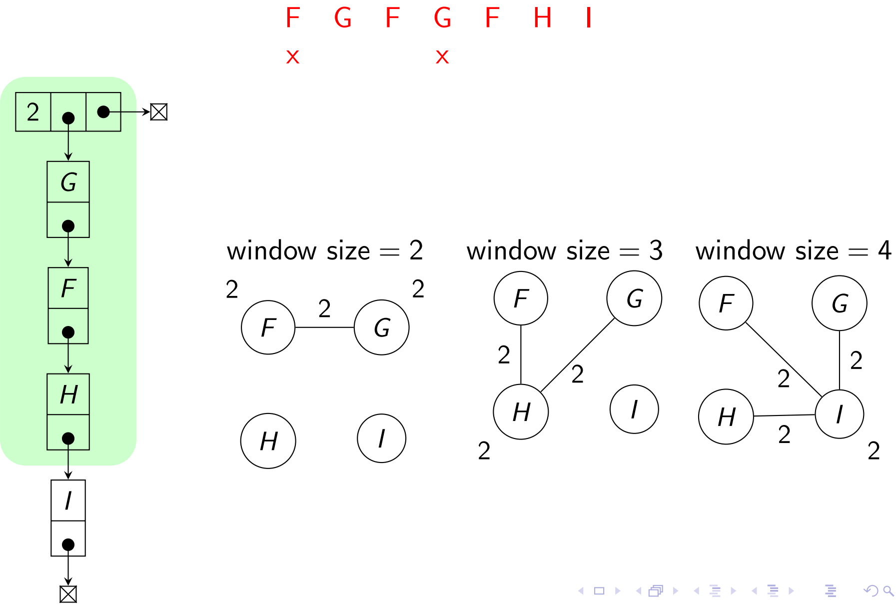
x x



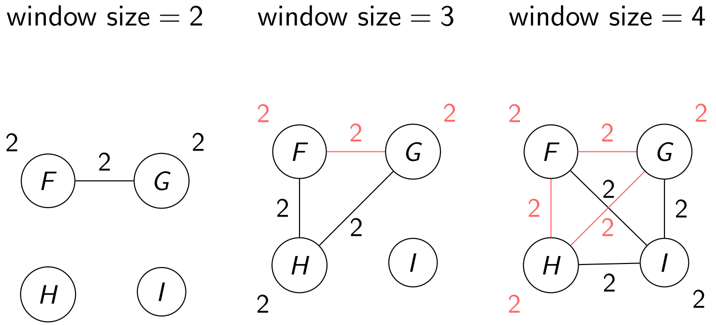
window size = 2 window size = 3



Window Growth



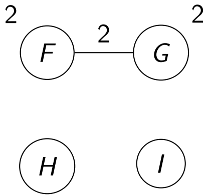
Accumulating Graphs



window size = 2 window size = 3 window size = 4

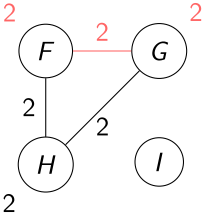
Computing Affinity

window size $= 2$



$$\frac{FG.freq}{\max(F.freq,G.freq)} = 1$$

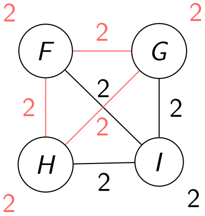
window size $= 3$



$$\frac{FH.freq}{\max(F.freq,H.freq)} = 1$$

$$\frac{GH.freq}{\max(G.freq,H.freq)} = 1$$

window size $= 4$

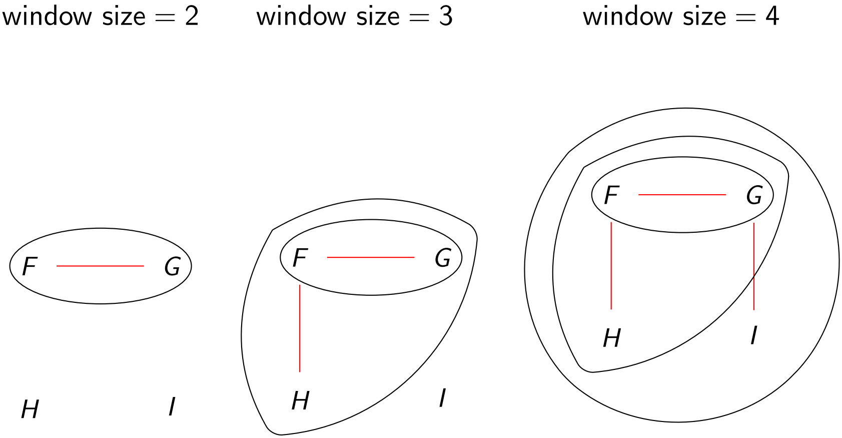


$$\frac{FI.freq}{\max(F.freq,I.freq)} = 1$$

$$\frac{GI.freq}{\max(G.freq,I.freq)} = 1$$

$$\frac{HI.freq}{\max(H.freq,I.freq)} = 1$$

Computing the Affinity Hierarchy



window size = 2

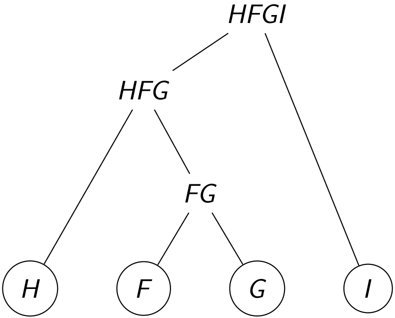
window size = 3

window size = 4

Computing the Affinity Hierarchy

F G F G F H I

x x



Time Complexity

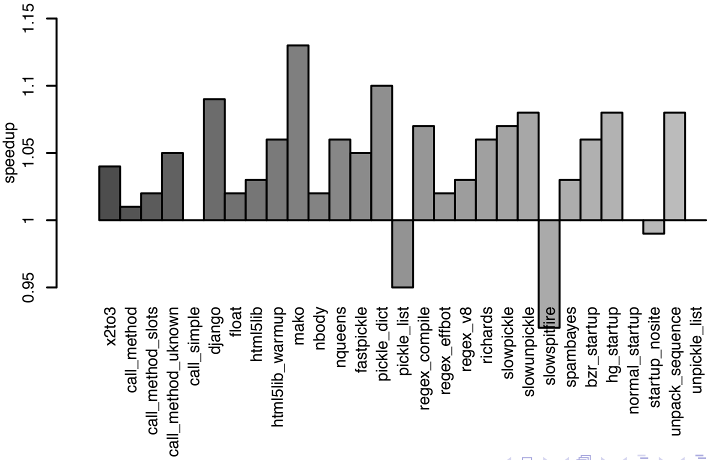
- The algorithm runs in time $O(\delta L W^2)$ in the worst case.
  - δ : Sampling rate
  - L : Length of the trace
  - W : Maximum window size
- In practice it performs much better.
  - Higher sampling rate leads to bigger partial window lists.

Implementation

- We implemented this algorithm within an LLVM compiler pass.
- To reduce the profiling cost, we use two threads
  - Analyzer thread: Analyses the window list and grows it.
  - Updater thread: Updates the frequency counts.

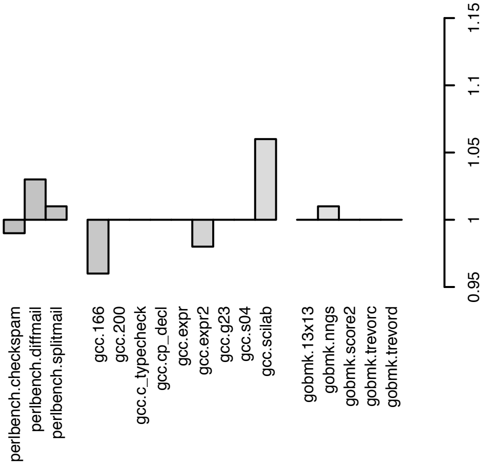
Speedup Evaluation: Python

- Speedup results for Python (Google's unladen swallow benchmark)
- Results are for sampling rate 0.001 and window size limit 15.
- The interpreter has been trained with *django*.



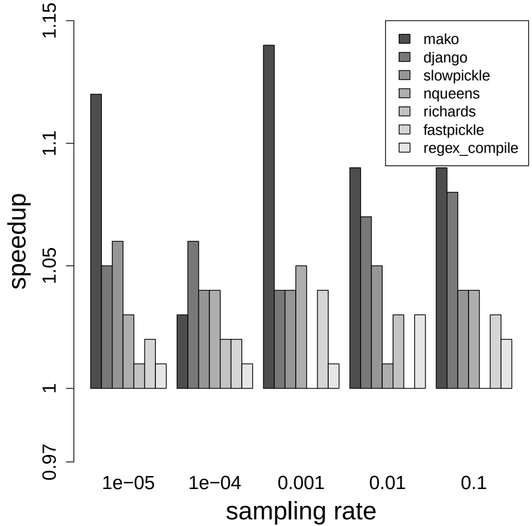
Speedup Evaluation: SPEC2006

- Speedup results for SPEC2006 (Perl, GCC, and Go)
- The programs have been trained with the provided training input.



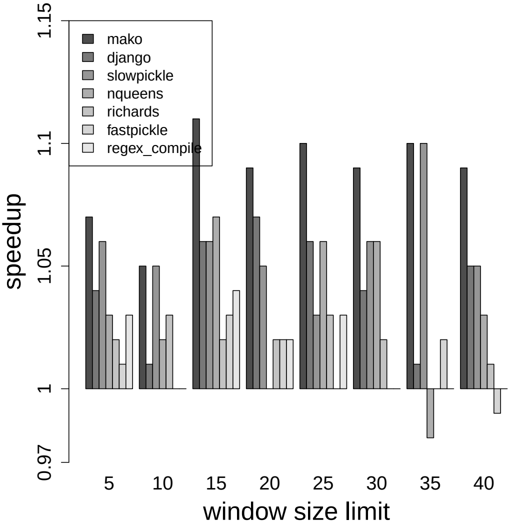
Sensitivity to Parameters

- Speedup sensitivity with respect to the sampling rate



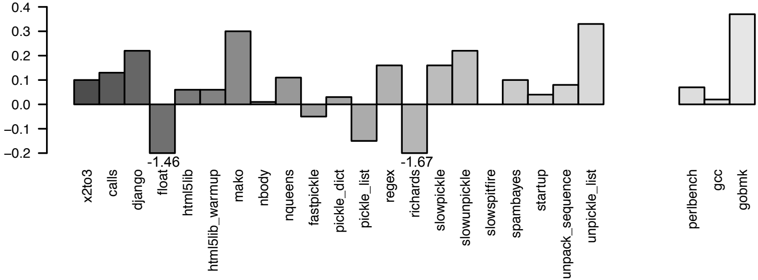
Sensitivity to Parameters

- Speedup sensitivity with respect to the window size limit



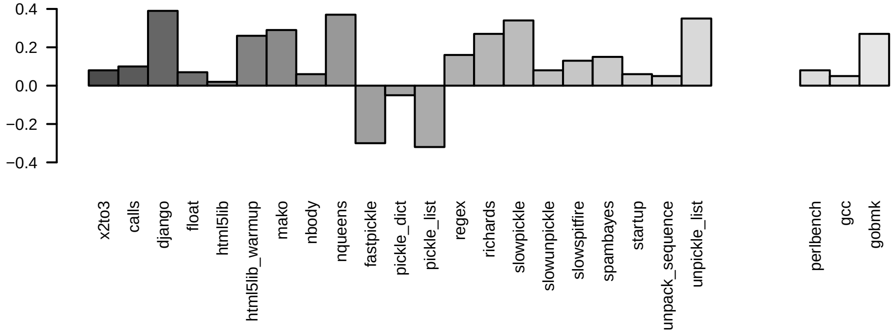
Evaluation

- Reduction in L1 instruction cache misses



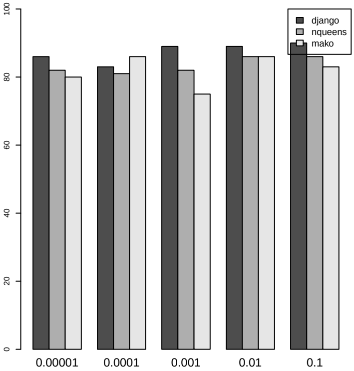
Evaluation

- Reduction in L2 cache (instruction) misses



Profiling Cost

- Using two threads (analyzer and updater) significantly reduces the profiling cost.
- The profiling cost is almost independent of the sampling rate.



Summary

- We presented a new efficient algorithm for exploiting reference affinity.
- It combines the affinity information in all window sizes and all affinity thresholds.
- We found our algorithm effective at improving the performance of Python interpreter,
- and to a lesser extent the Perl interpreter.
- The optimization does not cause significant slowdowns.
- It is robust across different parameterizations of the algorithm.

Thank You! Any Questions?