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?
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)
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

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

- Defined for every window size, as:

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

  \[
coco(F, H) = \frac{1}{\max(4, 1)} = 1/4
\]
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
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
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
× ×
Window Creation at Sampling Point

1  1

F G F G F H I
X  X

1

1

F

G

H

I
Window Growth

F G F G F H I
x x

1

F

G

1

1

F G

H I

1
Attempt for Window Growth

\[ F \quad G \quad F \quad G \quad F \quad H \quad I \]

\[ \times \quad \times \]

\[ 1 \]

\[ F \]

\[ G \]

\[ F \]

\[ \times \]

\[ 1 \quad 1 \]

\[ F \quad 1 \quad G \]

\[ F \quad H \quad I \]
New Window creation at Sampling Point

\[
F \quad G \quad F \quad G \quad F \quad H \quad I
\]

\[
\begin{array}{ccc}
1 &\rightarrow & 1 \\
F &\rightarrow & G \\
G &\rightarrow & \\
\end{array}
\]

\[
1 \quad 1 \quad 2
\]

\[
F \rightarrow G \\
H \rightarrow I
\]
No Window Growth (Cleanup)
Window Growth

\[
\begin{array}{c}
\text{F} \quad \text{G} \quad \text{F} \quad \text{G} \quad \text{F} \quad \text{H} \quad \text{I} \\
\times \quad \times
\end{array}
\]
Cleanup Record

\[ \text{Cleanup Record} \]

\[ \begin{array}{ccccccc}
\text{F} & \text{G} & \text{F} & \text{G} & \text{F} & \text{H} & \text{I} \\
\times & \times & \\
\end{array} \]

\[ \begin{array}{ccc}
1 & 1 & 1 \\
\times & \times & \\
\end{array} \]

\[ \begin{array}{c}
\text{G} \\
\text{F} \\
\end{array} \]

\[ \begin{array}{cc}
2 & 2 \\
\text{F} & \text{G} \\
\text{H} & \text{I} \\
\end{array} \]

Chen Ding, Rahman Lavaee, Pengcheng Li (UR)

ABC-Optimizer: An Affinity-Based Code Layout Optimizer

November 19, 2013 18 / 36
Cleanup Window and Add Window Counts

\[ F \quad G \quad F \quad G \quad F \quad H \quad I \]

\[ \times \quad \times \]

\[ 2 \]

\[ G \]

\[ F \]

\[ H \quad I \]

\[ 2 \quad 2 \quad 2 \]

\[ F \quad G \]

\[ H \quad I \]
Window Growth

\[FGFGFHI\]

\[\begin{array}{c}
2 \\
G \\
F \\
H
\end{array}\]

window size = 2

\[\begin{array}{c}
2 \\
F \\
G
\end{array}\]

window size = 3

\[\begin{array}{c}
2 \\
H
\end{array}\]
Window Growth

window size = 2

window size = 3

window size = 4
Accumulating Graphs

window size = 2

window size = 3

window size = 4
Computing Affinity

window size = 2

\[
\frac{FG.\text{freq}}{\max(F.\text{freq},G.\text{freq})} = 1
\]

window size = 3

\[
\frac{FH.\text{freq}}{\max(F.\text{freq},H.\text{freq})} = 1
\]

\[
\frac{GH.\text{freq}}{\max(G.\text{freq},H.\text{freq})} = 1
\]

window size = 4

\[
\frac{FL.\text{freq}}{\max(F.\text{freq},I.\text{freq})} = 1
\]

\[
\frac{GI.\text{freq}}{\max(G.\text{freq},I.\text{freq})} = 1
\]

\[
\frac{HI.\text{freq}}{\max(H.\text{freq},I.\text{freq})} = 1
\]
Computing the Affinity Hierarchy

window size = 2

window size = 3

window size = 4

\[ F \quad G \]

\[ H \quad I \]

\[ F \quad G \]

\[ H \quad I \]

\[ F \quad G \]

\[ H \quad I \]
Computing the Affinity Hierarchy

F G F G F H I
X

HFGI

HFG

FG

H F G I
The algorithm runs in time $O(\delta LW^2)$ in the worst case.

- $\delta$ : 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.
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

![Graph showing speedup sensitivity with respect to the sampling rate for various benchmarks.](image-url)
Sensitivity to Parameters

- Speedup sensitivity with respect to the window size limit

![Graph showing speedup sensitivity with respect to the window size limit for different programs like mako, django, slowpickle, nqueens, richards, fastpickle, and regex_compile. The x-axis represents the window size limit, and the y-axis represents the speedup. The graph indicates how the speedup varies with the window size limit for each program.]
Evaluation

- Reduction in L1 instruction cache misses
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.
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?