ABC Optimizer: Affinity Based Code Layout Optimization

Rahman Lavaee

Chen Ding

The University of Rochester
Dept. of Electrical & Computer Engineering
Computer Science Department †
Rochester, NY 14627

Technical Report TR-987

January 2014

Abstract

Modern software often has a large amount code and a dynamic execution pattern. Examples include interpreters and libraries for dynamic languages such as Python and Perl. It is interesting and important to study techniques to optimize code layout for interpreters and other large applications.

This paper presents a new technique for affinity based code layout optimization. By analyzing the reference trace of an interpreter, affinity optimization finds program functions that have affinity, i.e. often used together, and recognizes the code layout to group these functions together in memory. Algorithmically, the new technique is more general and efficient. The paper evaluates it by optimizing the Python interpreter and 5 programs from SPEC2006. Experiments show performance improvements on modern processors, sometimes in double digit percentages.

The research is supported by the National Science Foundation (Contract No. CNS-1319617, CCF-1116104, CCF-0963759), IBM CAS Faculty Fellowship and a grant from Huawei. Any opinions, fndings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

1 Introduction

Dynamic languages such as Perl, Python, JavaScript and MATLAB are increasingly popular. For example, Python is standard in almost all Linux distributions. A good reason is that most Linux installer programs are written in Python. Many other Python programs are in production use for purposes both commercial, e.g. Google, Yahoo, and scientific, e.g. CERN and NASA. A large number of widely used libraries, including scientific simulation and 3D animation, are written in Python. It is also used in programming new platforms including One Laptop per Child and Raspberry Pi. All of these uses require a Python interpreter, making it one of the most often executed programs on modern computers.

In this paper, we study the problem of code layout optimization for large dynamic applications such as interpreters. There pose unique challenges.

First, modern software are often bundled or installed with the code from many open-source libraries. Because of the large code size, an execution may incur as many misses for loading instructions as it does for loading data. Instruction misses may happen not just in the level-one (L1) instruction cache but also in the unified cache at lower levels and in TLB. Second, in the case of interpreters, the dynamic features such as dynamic typing, meta-programming and run-time inspection make traditional compiler optimization (Cooper and Torczon, 2010) less effective. The suitability of profiling analysis is questionable. On the one hand, the behavior of the interpreter may change significantly from script to script or between different inputs of the same script. On the other hand, there may be substantial commonality in the interpreter execution, since all scripts use the common functions such as parsing and basic control flows.

To solve these problems, we present affinity-based code layout optimization (*ABC-optimization*). It finds program functions that are used together (i.e. they have reference affinity (Zhang et al., 2006)) and places them together in memory. Reference affinity has previously been studied for optimizing the data layout. We adapt it to target code layout.

We generalize the previous algorithms, which analyze affinity in all footprint sizes (Zhong et al., 2004; Zhang et al., 2006). The affinity is either strict, i.e. always accessed together, or partial but for a fixed, pre-determined *affinity threshold*.

We describe a new algorithm that analyzes all affinity thresholds as well as all footprint sizes. We call the generalized algorithm *2D affinity analysis*. While the previous work finds a single hierarchy, 2D analysis finds a two dimensional matrix containing multiple overlapping hierarchies.

2D affinity analysis has several advantages for optimizing code layout. First, program functions are connected by calling relations and have variable affinities with each other. 2D affinity captures the variation. Second, 2D affinity analysis has the same asymptotic complexity as the previous work, while it obtains more accurate information. Third, in practice, the analysis is amenable to sampling, so the cost can be further reduced by reducing the sampling rate.

We have implemented the ABC-optimizer in the LLVM compiler and tested it on a Python interpreter and five SPEC INT benchmarks that have a large code size including a Perl interpreter and a version of the GCC compiler. For these programs, the ABC-optimizer reorders thousands of functions. We report the performance on a real system.

2 Reference Affinity

In this section, we introduce the preliminary concepts of the theoretical model of reference affinity. Every program's data access behavior can be represented by a data access trace. A *data access trace* is a sequence of data element accesses indexed by logical time. The *footprint distance* between two data element accesses at times t_1 and t_2, is defined to be the cardinality of the set of data elements accessed between (and including) t_1 and t_2.

Zhong et al. (Zhong et al., 2004) defined the concept of reference affinity groups for data elements on an access trace. A set of data elements belong to the same affinity group if they are always accessed close to each other. They formalized the notion of "close access" via a special relation called k-linked-ness, where the link length k acts as an affinity parameter.

Their definition will deliver a unique partition of data elements for every link length. Furthermore, it will impose a hierarchical structure over the affinity groups. That is, the affinity groups for link length k form a finer partition of the affinity groups for link length $k+1$, for every $k \geq 0$.

However, despite of these theoretical properties, this model suffers from strictness. In the sense that a small deviation from affinity may cause data elements to postpone forming a reference affinity group until larger link lengths.

For example, suppose the program's data access trace is the following.

$$ABABABABABXYXUVUA$$

Clearly, the affinity partition for zero link length consists of singleton groups. For link length 1, data elements form the affinity groups $\{A\}$, $\{B\}$, and $\{X, Y\}$, and $\{U, V\}$, and finally, with link length 2, they form the trivial partition $\{A, B, X, Y, U, V\}$. This way, the last reference to A has obscured the high reference affinity between A and B, which is unfortunate for optimization.

Weak reference affinity (Zhang et al., 2004) solves this problem by introducing a second parameter, affinity threshold. Like the strict version, weak reference affinity groups form unique partitions, for every pair of affinity threshold and link length. These partitions also adhere to a hierarchical structure with respect to both parameters.

Zhang et al. (Zhang et al., 2006) discover that neither strict reference affinity, nor weak reference affinity can efficiently be computed. Thus they propose a heuristic approach for the affinity problem. This approach basically samples windows of a certain footprint size (which needs to be carefully chosen) and takes every window which contains references to both x and y as a certificate for reference affinity between x and y. If the number of certificate windows for x and y reaches a confidence threshold, x and y will be declared as a pair with affinity relation. The estimated affinity groups will finally be computed by taking the transitive closures of the pairwise affinity relation graph.

As for the parameters governing this approach (i.e. maximum footprint size and confidence value), they suggest using $2gk$ as the window size, where k is the link length and g is an upper bound estimate on the size of the affinity groups. This choice guarantees that for every two elements x and y which belong to the same affinity group, a sampled window covers both x and y with probability 0.5 given it covers x. Thus when setting the confidence value to 0.5, the sampling algorithm will, in expectation, successfully discover all the strict affinity relations. Analogously, for weak affinity groups, setting the confidence value to $\theta/2$, where θ is the affinity threshold, will guarantee discovering all the weak affinities in expectation.

The authors also propose an algorithm to compute a single level of reference affinity (a single footprint size). The average time complexity of their algorithm is $O(L\delta\omega^2 + L\delta\pi)$ where L is the length of the trace, δ is the sampling rate, ω is the maximum footprint size, and π is the average time length of ω-sized footprint windows [1]. In order to compute affinity for all footprint sizes up to ω, a naive approach requires performing their single level analysis for every footprint size which makes the running time depend cubically on ω.

To solve this problem and to keep the time complexity depend quadratically on ω, they propose varying the footprint size in a logarithmic scale.

In contrast, we design a streaming algorithm that computes reference affinity for all footprint sizes up to the given maximum footprint size (footprint limit), while it achieves the average time complexity of

[1] The analysis in (Zhang et al., 2006) is inaccurate. The authors claim that their algorithm runs in time $O(L\delta\omega^2)$ and completely ignore the cost of growing windows. In fact every time a new element of the trace is analyzed, we need to examine all the currently growing windows for possible growth. An essential part of our algorithm deals with reducing this cost.

$O(L\delta\omega^2 + L\omega)$ (which may even be smaller than the complexity of the single level algorithm mentioned above).

Furthermore, we will show that the variation of our algorithm that limits the affinity computation to footprint sizes on a logarithmic scale, runs in time $O(L\omega)$.

3 Affinity-Based Code Layout

This section presents the ABC-optimizer design. It is profiling-based and has three steps: trace collection, affinity analysis and function-level reordering. Affinity analysis is the most complex step and has several components: sampled window analysis, window counting, co-occurrence graph construction, and coalescing of affinity hierarchies.

3.1 Instrumentation and Trace Collection

Affinity-based reordering may be performed at function or basic-block granularity. Function reordering is easier to implement in a compiler, and the optimization is more portable across compilers and target platforms. Modern applications such as interpreters typically have many functions, and most of them are small. Therefore, for code layout optimization, function reordering is challenging and has sufficient potential for improvement. In addition, we may collect a function-level execution trace more efficiently, as we show next.

A function execution trace is an instruction trace where each instruction is replaced by the function that contains the instruction. For affinity analysis we remove all consecutive appearances of the same function. We call it the *trimmed function trace*. The goal of the instrumentation is to collect the trimmed function trace with the minimal cost.

We show two intermediate solutions before describing our solution. The first is to instrument at the start of each function. When a function F is called, the instrumentation covers the subsequent execution until F calls another function G. The solution is efficient but incomplete. Once F finishes calling G, the return from G to F is not recorded.

The second solution is to instrument at the start of every basic block. It solves the completeness problem of the first solution. When G is returned, it will execute the next basic block, which will record F. Instrumentation at every basic block, however, is often unnecessary. If a function executes consecutive basic blocks, e.g. in a loop, without calling another function, it is unnecessary to record the execution of the same function multiple times.

Our solution instruments at two places: the entry of each function, and in each function, the return point after every call site. The solution is both complete and efficient. It is complete because all function call entries and returns are captured. It is efficient because there is no consecutive recording of the same function execution.

3.2 Sampled Footprint Window

For each pair of functions, the algorithm computes their affinity by counting their joint appearances in *sampled footprint windows*. A *footprint window* is distinguished by two attributes. The first is the starting (time) point in the trace. The second is the size of the footprint. Two footprint windows differ if and only if they differ by at least one attribute. A footprint window is *sampled* if it starts from a sampling point. We will count only sampled windows.

To illustrate, in the short trace below,

```
F   G   F
x
```

we sample the first F, as marked by an underlying 'x', but not the next two, G, F. In this trace, we have three non-unit size time windows, FG, GF and FGF, but only two footprint windows, because FG and FGF represent the same footprint window, i.e. same starting point and same footprint. Of the two footprint windows, only one is sampled.

To be less verbose, we shorten the term sampled footprint window to *sampled window*. Since we do not sample time windows, the slightly shorter term should not cause any confusion.

3.3 Counting All Sampled Windows

The goal of the algorithm is to check joint appearances in *all* relatively small sampled windows. Thus for every sampled starting point p, we will count all the windows starting from p with up to a certain footprint size (which we call the *footprint limit*).

Consider another example trace:

F	G	F	G	...	F	G	H
	x		x			x	

There are three functions executed in the trace. We sample three of the calls to G.

A reader can verify the understanding of our definition by finding all sampled windows in this trace. The right answer is that the trace has two sampled windows with footprint $\{F, G\}$, one sampled window with $\{G, H\}$, and two sampled windows with $\{F, G, H\}$.

The example shows proportional information obtained through sampling. In the last example, G occurs more often with F than with H. Our sample counts capture this difference because there are two G, F windows and only one G, H window. With uniform sampling, window counting should discover different levels of affinity in the execution.

As we noted, we limit the maximal footprint size w so as not to analyze windows that are too large. In the example, if we set the footprint limit $\omega = 2$, then the two windows with $\{F, G, H\}$ are too large and will not be counted.

In order to count all windows, a naive solution is to create one growing window for every sampling point and incrementally grow that window until it reaches the footprint limit. However, this solution causes repeated trace processing if a sampled window contains other sampling points. Next we present a one-pass algorithm that counts all sampled footprint windows.

3.4 Single-Pass Window Counting

The top-level procedure for the algorithm is given in Algorithm 1. For each function invocation in the execution trace, the analyzer calls this procedure and passes the identity of the called function.

Algorithm 1

```
1: procedure RECORD_FUNC_EXECUTION(fid)
2:     added ←ADD_NEW_ELEMENT
3:     if added then
4:         GROW_WINDOW_LIST
5:     end if
6: end procedure
```

The algorithm starts from the first sampling point and gradually includes subsequent points in the trace. Effectively, it grows a footprint window to include all footprint windows up to the footprint limit ω. We may call each footprint window a growing window until the footprint size reaches the limit ω.

The algorithm stores the relevant trace (the trace that contains all the sampled windows) in the form of a two-level doubly linked list. Let us call the upper-level list the *window list* and the lower-level lists *function lists*. Every element in the window list is a partial window, which consists of a window count along with a list of function ids (function list).

Consider the following trace which has five function execution records. Figure 1 gives a visual view of its analysis that we will use as a running example to explain the algorithm.

F	G	F	G	F
x			x	

After the first call, we have the first node, representing F. Since F is assumed to be a sampling point, Algorithm 2 (add_new_element) creates a partial window containing a single function record F, and assigns its window count to one, which means that there is one function list containing F. The algorithm inserts this partial window at the end of the currently empty window list. This step is shown in Figure 1(a). Now we just need to update the individual frequency for function F.

Algorithm 2

```
1: procedure ADD_NEW_ELEMENT(fid)
2:     wCount ← 1 with probability s and 0 with probability 1 − s
3:     if wCount = 1 then
4:         window =CREATEEMPTYWINDOW
5:         window.count = 1
6:         windowList.PUSHBACK(window)
7:     end if
8:     if ¬windowList.EMPTY then
9:         funcRec ← CREATEFUNCTIONRECORD(fid)
10:        windowList.BACK.PUSHBACK(funcRec)
11:        return True
12:    else
13:        return False
14:    end if
15: end procedure
```

In the second step, since this execution of G is not sampled, we don't create a new partial window and simply insert the record at the end of the only function list, as shown in Figure 1(b). Now it's time to update the affinity information, i.e. the individual and joint frequencies. The only existing partial window has grown by including a new function record G. Thus Algorithm 3 (grow_window_list) passes the complete window list to Algorithm 4 (update_affinity).

Let us explain this algorithm in more detail. Nominally, each partial window contains a portion of the relevant trace, while conceptually, it also covers all the records owned by the partial windows on its right. Let us call this latter abstract notion of the window the *effective window*. To be less verbose, throughout this section, we may simply use the term "window" to refer to a partial window. Every effective window that has grown with the newly inserted element will contribute to the individual and joint frequency of that element. Algorithm 3 computes this portion of the window list (which we call the grown window list) and passes it to the affinity update procedure. The affinity update procedure (Algorithm 4) walks over this grown window list and for each effective window it first increments the individual frequency of the newly inserted record (G) by its window count. Then, walking over the function records in the corresponding (partial) window, it updates the joint frequencies between those records and the newly inserted record. Similar to the individual frequencies, for every function record, all the (grown) effective windows on the left hand side of the current window (including itself) contribute to the joint frequency.

In the current step, there is only one window which by itself is the only effective window. So the affinity update procedure increments $singleFreq\langle G\rangle[2]$ by one. Because the size of the effective window is 2 and its window count is 1. To update the joint frequencies, it walks over the window and encounters F (we ignore

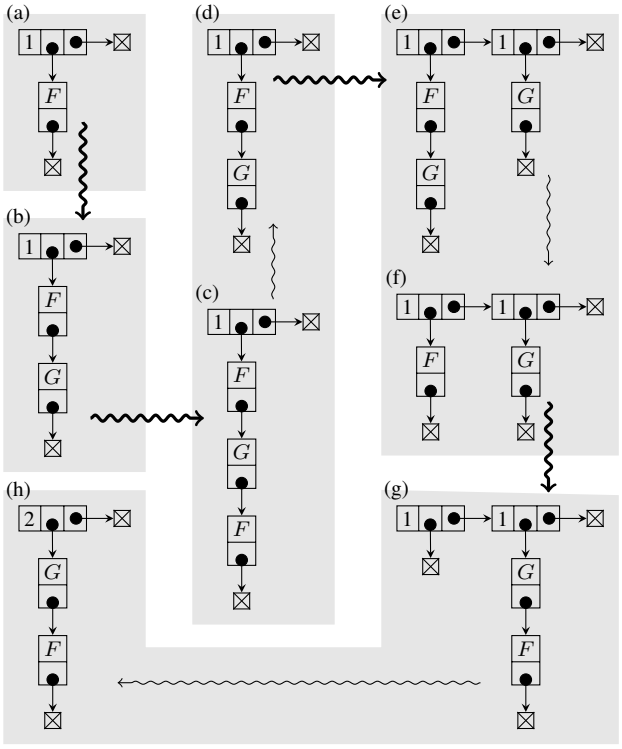

Figure 1: Single-pass window counting for the "FGFGF" example (in which the first F and the second G are sampled points). Each shaded box shows the content of the two-level linked list when processing each element. After the first element, we have one partial window with count 1 and function F. After the last element, we have one partial window with count 2 and functions F, G. The analysis finds the two sampled windows with footprint $\{F, G\}$ in a single pass over the trace.

updating the joint frequency of $\langle G, G\rangle$). Then it increments $jointFreq\langle F, G\rangle[2]$ by one. The computation finishes now because there are no other grown windows

Our algorithm ensures that upon completion of recording every function execution, the window list will contain at most one instance of every function. This will guarantee our efficient single pass window counting. So far in the trace, we didn't encounter a duplicate function. However, in the third step, we encounter F when the current window contains a record of the same function. First we insert the record into the window list. The result is shown in Figure 1(c). Algorithm 3 then discovers that F had already existed in the window list. Furthermore, it can tell exactly where that function record is, by keeping tracking of a pointer to that record, and another pointer to its containing window. No effective window has grown and we don't need to update the affinity information. Thus the algorithm continues by deleting the old record and updating the pointers accordingly. Finally, we end up with the window list shown in Figure 1(d).

Basically, in the case where we encounter a duplicate function record, the windows that grow are precisely the ones appearing on the right side of the window that contained the old function record. So far, the window list contained only one window. But the fourth step is a sampling point. So we will insert a new window containing G at the end of the window list as shown in Figure 1(e). Algorithm 3 then passes the portion of the window list on the right side of the window that contained the old function record to the affinity update procedure. This portion consists of the only window that has grown which is the newly inserted window (in fact it is born). Finally since G was a duplicate record, we delete the old record and end up with the window list show in Figure 1(f).

Now what would happen if by deleting the old function record, its containing window becomes empty? This is the trickiest part of the algorithm which happens in the final step when we encounter F again. We first insert F into the window list. Then we need to pass the grown portion of the window list to Algorithm 4 and update the affinity information. The result of this procedure is incrementing both $singleFreq\langle F\rangle[2]$ and $joinFreq\langle F, G\rangle[2]$ by one. After returning from this procedure we delete the old record of F in the left window. The result after these steps is shown in Figure 1(g).

As we can see, the leftmost window has shrunk to an empty list while its window count is still one. However, as we mentioned earlier, its corresponding effective window covers exactly the functions appearing on its right hand side. So effectively, this window is the same as the one on its right. So we can simply add its window count to the window on its right and then safely remove it. The final window list is shown in Figure 1(h).

Notably, since the footprint size is bounded by a footprint limit, we will remove every window once it grows to the footprint limit ω. This is accomplished through lines 5 to 10 of Algorithm 3. If the size of the window list overflows the limit, the right window to be deleted is the oldest one, which is at the beginning of the window list.

3.5 Parallel Window Counting

Our algorithm guarantees that the number of partial windows in the window list is proportional to the sampling rate. As a consequence the nested loop in Algorithm 4 incurs less overhead. In order to further reduce this overhead, we run Algorithm 3 and Algorithm 4 using the *producer-consumer* framework.

Initially, we spawn one thread for each algorithm, which we respectively call *the analyzer* and *the updater*. Whenever the analyzer thread discovers a growth it inserts a copy of the grown portion of the window list into a lock free queue. The updater thread takes every update entry (a grown window list) from the queue and performs the required updates. Without any locking, we can periodically count the number of elements in the queue and spawn a new updater if this number goes beyond a threshold.

This modification significantly reduces the profiling cost for higher sampling rates, as we will show in evaluation.

Algorithm 3

```
1: procedure GROW_WINDOW_LIST(fid)
2:     if windowPtr[fid] = null then
3:         totalSize ← totalSize + 1
4:         UPDATE_AFFINITY(windowList)
5:         if totalSize ≥ ω then
6:             oldestWindow ← windowList.FRONT
7:             totalSize ← totalSize − oldestWindow.SIZE
8:             oldestWindow.CLEAR
9:             windowList.REMOVE(oldestWindow)
10:        end if
11:    else
12:        grownList ← windowList.SUBLIST(windowPtr[fid])
13:        UPDATE_AFFINITY(grownList)
14:        windowPtr[fid].REMOVE(funcRecPtr[fid])
15:        if windowPtr[fid].EMPTY then
16:            nextWindow = windowPtr[fid].NEXT
17:            nextWindow.count += windowPtr[fid].count
18:        end if
19:    end if
20:    windowPtr[fid] = windowList.BACK
21:    funcRedPtr[fid] = windowList.BACK.BACK
22: end procedure
```

Algorithm 4

```
1: procedure UPDATE_AFFINITY(fid, grownWindowList)
2:     window ← grownWindowList.BACK
3:     ℓ ← 0
4:     while window ≠ null do
5:         ℓ ← ℓ + window.SIZE
6:         singleFreq⟨fid⟩[ℓ] += window.count
7:         funcRec ← window.BACK
8:         while funcRec ≠ null do
9:             effWin ←= window
10:            ℓ2 ← ℓ
11:            while effWin ≠ null do
12:                jointFreq⟨fid, funcRec.id⟩[ℓ2] += effWin.count
13:                effWin ← effWin.prev
14:                ℓ2 ← ℓ2 + effWin.SIZE
15:            end while
16:            funcRed ← funcRec.prev
17:        end while
18:        window ← window.prev
19:    end while
20: end procedure
```

Algorithm 5

```
1: procedure CUMULATE_FREQ_COUNTS
2:     for ℓ ← 2 to ω do
3:         for each fid do
4:             singleFreq⟨fid⟩[ℓ] += singleFreq⟨fid⟩[ℓ − 1]
5:         end for
6:         for each pair ⟨fid1, fid2⟩ do
7:             jointFreq⟨fid1, fid2⟩[ℓ] += jointFreq⟨fid1, fid2⟩[ℓ − 1]
8:         end for
9:     end for
10: end procedure
```

3.6 Rolling Up the Frequency Counts

At every step, we add an element to the end of the list. We remove the element with the same function id if the element already exists in the list. The addition may grow some sampled windows and leave other windows unchanged.

For every window growth, the algorithm updates the individual and joint frequencies. Assume that function f has been added to window W, resulting in the larger window $W \cup \{f\}$. We increment the individual frequency of f for footprint size $\|W\| + 1$. If this window grows again, the frequency of f in the larger window needs also an increment. In order to save computation time, we assume that every window will eventually grow to the footprint limit. Thus we only update the frequency for the current footprint size. At the end, we will cumulate the frequencies in a single pass.

For the joint frequency, we increment by the window count for pairings between f and each $a \in W$, for footprint size $\|W\| + 1$. Our aforementioned assumption is used again, which helps us similarly to defer the increments for larger footprint sizes to the end of the analysis. Nonetheless, this stage remains the costliest stage in our analysis. To further reduce the cost, we will store all joint frequencies of a pair in a single array. As we traverse the trace list, we update contiguous elements in an array.

Once the program terminates, we have all the partial frequency counts (individual and joint). Now we need to cumulate these counts for all footprint sizes in increasing order. This stage is shown in Algorithm 5.

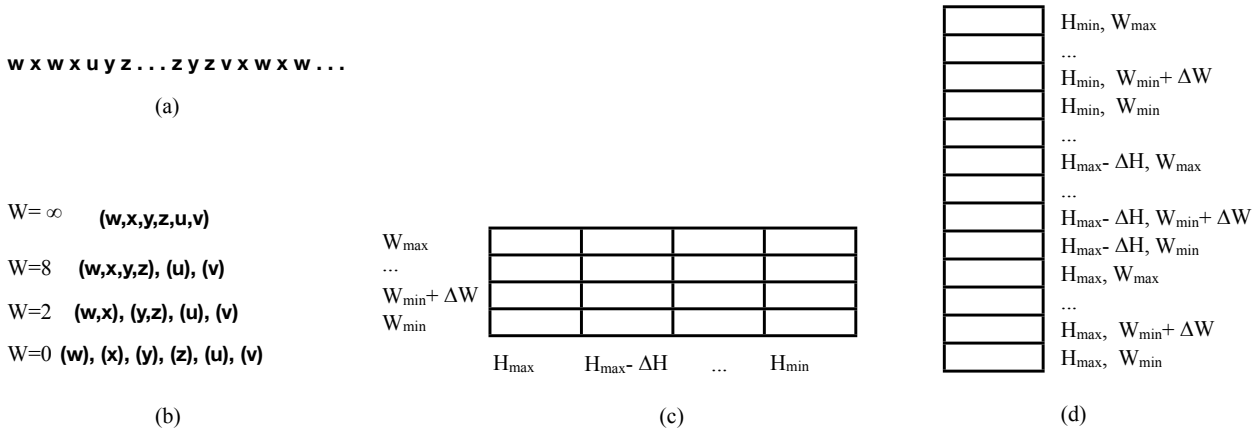

Figure 2: Two-dimensional reference affinity analysis. (a) An example execution trace executing 6 functions. (b) Strict affinity hierarchy ($H = 100\%$), which is a single hierarchy over different window lengths W. (c) Two-dimensional affinity matrix over different window lengths W in columns and different affinity thresholds H in rows. Each column (bottom up) is a single hierarchy, so is each row (left to right). (d) Coalescing the two-dimensional hierarchies into a single hierarchy.

3.7 Coalescing Affinity Hierarchies

In the past, reference affinity had a single hierarchy. In this section, we first give an example illustration and then describe the extension to the two-dimensional hierarchies and the method to coalesce these hierarchies to produce the optimized code layout.

Figure 2(b) shows the strict reference affinity hierarchy for the execution trace in Figure 2(a). In (a), "..." means functions other than those given in the trace. The 0 is shown in (b). Each level is labeled by a footprint size W. When $W = 1$, there is no co-occurrence, so each element is in its own affinity group. As W increases, functions w, x and y, z join the same group. When W is infinite, all functions come into a single group. We see that each level is a partition of all functions, and each lower level is a finer partition of the upper level. These properties were proved in (Zhong et al., 2004).

We call each partition on the same level an *affinity partition*. The hierarchy in Figure 2(b) has 4 affinity partitions.

In our algorithm, we compute the affinity groups using the co-occurrence graph. Using the edge and node weights, we check which pairs have affinity. Conceptually, only these edges remain in the graph after checking. Each connected sub-graph is then an affinity group.

Partial affinity can be easily tested. Given a threshold H, two nodes, A, B, have partial affinity if and only if

$$\frac{AB.freq}{\max(A.freq, B.freq)} \geq H$$

For a footprint size W and a threshold H, we have an affinity partition. By iterating over all W between W_{min} and W_{max} at increment ΔW and all H between H_{min} and H_{max} at increment ΔH, we find all affinity partitions. These partitions fill a two-dimensional matrix as shown in Figure 2(c), which we call an *affinity matrix*.

The affinity matrix contains affinity hierarchies in two dimensions. Each column is an affinity hierarchy, from bottom up as the footprint size grows. Each row is also an affinity hierarchy, from left to right as the threshold decreases.

In a complete analysis, we have $W = 1, \ldots, \infty$ and $H = 1, \ldots, 0$. At lower left of the affinity matrix, we have $W = H = 1$, and each element is in its own affinity group. At the upper left, we have the largest window and a single affinity group including all functions. At the lower right, we have the 0 threshold and also a single affinity group including all functions. In the middle are the useful partitions that show how affinity relation expands for different footprint sizes and for different degrees of partial affinity.

For code layout optimization, we coalesce the affinity matrix into a single affinity hierarchy. First, we choose a linear order to traverse the matrix. Then during the traversal, we union each pair of successive affinity partitions so the merged partition is their smallest coarser partition.

In practice, we choose W not too large and H not too low. We use the linearization of the hierarchy matrix shown in Figure 2(d). Given the small steps in changing W and H, the affinity relation changes gradually. We expect that other linearization schemes would produce similar results as long as W and H choices are closely interleaved. We will measure the sensitivity to the parameters W, H.

Once the affinity matrix is coalesced into a single affinity hierarchy, the generation of the optimized function layout is simply a bottom-up traversal of the final hierarchy. This is the same as done in the past work (Zhang et al., 2006). However, that work missed a crucial point. Whenever we merge two linear partitions, we have four ways to join them. Inspired by (Pettis and Hansen, 1990), we join the two partitions in the way that gives the highest local reward, i.e., the highest affinity between the joining points.

3.8 Complexity

The key idea for analyzing the time complexity of ABC-Optimizer is that at every moment the number of elements in the trace list is bounded by the footprint limit (ω).

Let L be the length of the trace and δ be the sampling rate. Algorithm 3 uses a single iteration to copy the grown portion of the window list and will run in time $O(L\omega)$. This grown window list is then given to Algorithm 4 which updates the individual and joint frequencies through a nested iteration over this list. Since there are $L\delta$ sampled windows in average, this algorithm runs in average time $O(L\delta\omega^2)$. Thus overall, the profiling part of the algorithm runs in time $O(L\delta\omega^2 + L\omega)$ which beats the running time of the algorithm in (Zhang et al., 2006).

For the sake of completeness, let us also analyze the complexity of the postprocessing stage of our algorithm (i.e. cumulation and coalescing).

The cumulation procedure (Algorithm 5) runs in time $O(\omega N^2)$ where N is the total number of functions.

During this coalescing stage, we compute a $\omega \times \tau$ affinity hierarchy, where τ is the maximum threshold level. We use disjoint set forest structure to keep and update the affinity groups. First we need sort all the edges in the affinity graph based on our notion of affinity hierarchy which takes $O(N^2 \lg N \times \omega)$ running time. Then we update the disjoint set forest by considering the edges in the sorted order, which is accomplished in time $O(N^2)$. Finally, to computing the optimized layout, we traverse the partition tree of the disjoint set structure which takes $O(N)$.

Regarding the space complexity, as we noted, the size of the window list is bounded by $O(\omega)$, and there remains only the frequency counts, for which the algorithm allocates $O(\omega N^2)$ space, in the worst case.

We will later see that for function reordering, since the function size is relatively large, we can guarantee the best optimization results without requiring the footprint limit to be larger than about 10. This allows the analysis cost to less extensively depend on the footprint limit. However, we believe that our method can also be applied for data layout optimization and code layout optimization in a finer grain. For these applications, it is conceivable that a higher footprint limit would result in better performance. Thus it is worthwhile to mention that by computing the frequency counts on a logarithmic scale, we are able to get the worst case running time of $O(L\omega)$ for the profiling stage. To accomplish this, we need to slightly modify Algorithm 4 so that it only updates the frequency counts for windows of size 2^i with i being bounded by $\lceil \lg \omega \rceil$. The modification is as follows.

Before entering the outermost loop at line 4 we build an array of log scale window counts $LogWC$ that will contain at most $\lceil \lg \omega \rceil$ elements. We assign $LogWC[i]$ to be the number of effective windows with a size in the interval $(2^{i-1}, 2^i]$. We can build this array in time $O(\omega)$. While we are building $LogWC$, we also update the individual frequencies.

Now we replace the innermost while loop of the algorithm (lines 11 to 15) where we update the joint frequency, by an iteration over $LogWC$. The iteration starts from the index j where 2^j is the smallest power of two greater than or equal to ℓ, and for each index i we increase the joint frequency for footprint size 2^i by $LogWC[i]$.

Let us analyze the running time of the modified algorithm. The dominating part is the innermost loop in Algorithm 4. Thus let us count how many times we update the joint frequency. Each joint frequency update involves a pair of function records; an old one and a new one. So one way to count them is to compute how many times a single function record f occurs as the old element in this pair.

Just after f is inserted into the window list, and when the next function is inserted, f can occur in $\lg \omega$ frequency updates (as the old element). Recall that at every step of the algorithm, we insert a function record at the back of the window list, copy the grown window list, and send it to Algorithm 4. f can occur in a pair only if the grown window list includes f. Let us denote by k_t the number of function records after f at time t. If the grown window list contains f then $k_t = k_{t-1} + 1$ and crucially, the number of frequency update pairs f can occur in is bounded by $lg\omega - \lg k_t$. Furthermore, we know that k_t can increase up to ω (It may not reach ω if f gets removed by inserting a new identical function record into the list). A straightforward summation analysis shows that this adds up to 2ω. In other words, f may occur in at most 2ω frequency update pairs until it gets removed from the list.

Since at every step we insert a function record into the list, overall, we will have at most $O(L\omega)$ joint frequency updates.

We also stress that, although we used binary logarithmic scale here, to achieve a more accurate analysis, we can use $1 < b < 2$ as the base of the logarithm, with a multiplicative increase in cost. Generally, the cost of the algorithm is $O\left(L\omega(b-1)^{-2}\right)$ for every base b.

4 Evaluation

In this section we present our evaluation of ABC-optimizer on Python interpreter and 5 programs from SPEC2006 whith large source code. We report the improvement in performance and the reduction in Level 1 and Level 2 instruction cache miss count.

We also compare the performance of ABC-optimizer with the procedure reordering optimizer proposed in (Pettis and Hansen, 1990). This optimizer computes the call frequency graph and lays out the functions by considering the call graph edges in nonincreasing order of their weights. Similar to ABC-optimizer, when considering an edge, the corresponding lists of its endpoints are joined in the direction that has the highest local reward, i.e. the highest call frequency between the joining points. We call this optimizer call frequency based optimizer or CF-optimizer in short.

4.1 Compiler Implementation

The implementation of the ABC-optimizer includes a compiler and a profiler. For the compiler, we use LLVM (Lattner and Adve, 2005) and add two passes into the compiler. The first is instrumentation at function entries and call returns, as described in Section 3.1. Using the compiler, we assign a global identifier to each function and record the identifier in the instrumentation. To do so, we require a program to be compiled into a single LLVM bit-code file. During the instrumentation pass, we first iterate over all functions and assign an integer index as global identifiers.

When the instrumented program is run, it generates the trimmed function trace. The profiler analyzes the trace and identifies the affinity hierarchy, using the algorithm described in Section 3. It generates the optimized code layout as a permutation of function identifiers.

Finally the second compiler pass is for function reordering. This is done simply by recompiling the uninstrumented bit-code file.

4.2 Experimental Setup

Python 2.7.5 Interpreter The current stable Python release is 3.3.2 but to run our test benchmark suite (see the following), we are required to use version 2. We choose one of the latest which is 2.7.5. The interpreter is implemented in C and contains 2829 functions. Each permutation of these functions is a possible layout.

Our optimizer, in design and implementation, does not target Python in particular. In fact, we have optimized an Perl interpreter. But in the absence of a comprehensive test suite for Perl, we have not conducted a careful experimental study for Perl as we have done for Python.

Google Unladen Swallow Benchmark Suite Python is extensively used in Google's infrastructure and its hosted App Engine system. In 2009, Google engineers started the Unladen Swallow project to build a fast Python implementation. As part of the project, they compiled a set of Python programs for testing. Here we use them as the inputs to our Python interpreters. There are 32 benchmarks, including both IO intensive and CPU intensive workloads.

Unladen Swallow is to represent applications common in online and server processing. A test program, *django*, is named after the widely used Python web application framework of the same name. We use *django* as the training input for the ABC-optimizer.

SPEC 2006 Benchmarks Five of the integer benchmarks show a significant instruction miss rate on our test platform. The programs are: *perlbench*, which is a cut-down version of Perl version 5.8.7 including 7 third-party modules such as SpamAssassin and HTML-Parser; *gcc* version 3.2, *gobmk*, the GNU computer Go program, *povray*, a ray-tracer; and *xalancbmk*, which is a modified version of Xalan-C++, an XSLT processor for transforming XML documents to HTML. ABC-optimizer uses the training input provided by SPEC. We include the results for all the reference inputs.

Hardware Platform Our test machine has an E5520 processor, launched in Q1 2009, with 4 Nahalem-EP cores running at 2.26GHz. The instruction cache is 32KB with 32-byte lines and 4-way set associativity. The second level cache is 256KB, 64-byte lines, and 8-way set associative. The first two levels are private for each core. The third level cache is 8MB shared by all cores.

Performance Testing To measure time, we run the tests sequentially on an unloaded system. Both the Unladen Swallow and SPEC provide a test harness, which we use to record the average speedup (reported by Unladen Swallow harness) and the run time (reported by SPEC harness). The Python harness includes three options for the accuracy and in turn, duration of the run; *fast*, *regular*, and *rigorous*. We use *fast* for training and *rigorous* for testing. The SPEC harness runs the tests once by default.

4.3 Analysis of the Results

Figure 3(a) shows for each of the two optimizers, the performance speedup for 29 scripts of Python, 3 scripts of Perl, 7 inputs of GCC, 5 inputs of Go, and the single inputs of povray and xalancbmk, for a total of 46 tests on 6 programs. Let's study each program in detail.

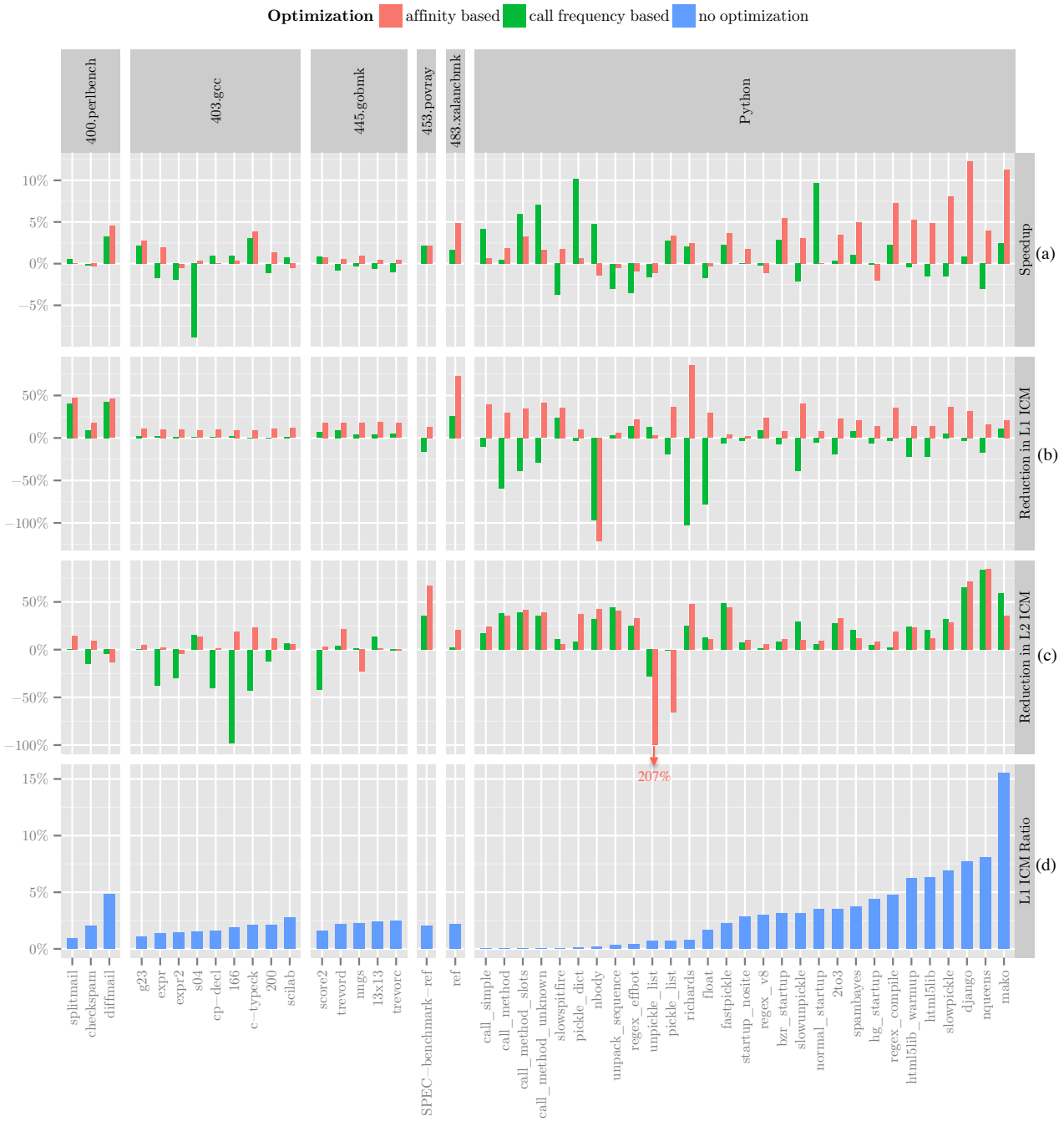

Figure 3: Performance improvement (a) and miss count reduction in Level 1 (b) and Level 2 instruction caches (c) (L1 ICM and L2 ICM) for both affinity based and call frequency based code layout optimizations, along with the base miss ratio for Level 1 instruction cache (d). (Left) 5 SPEC 2006 INT benchmarks running reference inputs. (Right) Python running 29 Unladen Swallow scripts. a total of 46 tests on 6 programs. The test inputs are sorted in the increasing order of their Level 1 instruction cache miss ratio.

Perl Both optimizers have a decent improvement over *diffmail*, with ABC-optimizer leading by about two percent. For the other two test inputs (*splitmail* and *checkspam*), the improvements or degradations are less than 0.6 percent and negligible. However, if we look at the reductions in L1 ICM and L2 ICM, in all but one case (L2 ICM for *diffmail*) ABC-optimizer has better impact on miss count. We can also notice that the ABC-optimizer performs better on the rightmost test input which has significantly larger L1 ICM ratio (5%).

GCC Our speedup improvements over CF-optimizer are much more significant. Among the 9 reference inputs, ABC-optimizer performs better on 6 inputs, and worse on the rest only by at most 1.4% difference in speedup. The miss count results are more interesting. For both optimizations, the reduction in L1 ICM are almost uniform across different inputs with our optimizer performing about 9% better. For the majority of the inputs, CF-optimizer significantly increases the L2 ICM, while for all but one of the inputs (*expr2*), ABC-optimizer decreases this value. This implies that for GCC, most of the speedup improvements are due to the decrease in L2 ICM.

Gobmk Out of the 6 inputs, CF-optimizer degrades performance for 5 inputs by at least 0.33%, while ABC-optimizer improves the performance by at least 0.42% for all inputs. Similar to GCC, for both optimizations, the reduction in L1 ICM is almost uniform across different inputs, again with ABC-optimizer leading by about 10%. However, there seems to be no meaningful comparison between the optimizations in terms of the reductions in L2 ICM, except that ABC-optimizer performs slightly better in average.

Povray For both optimizers, the speedup improvement over the single input is about 2%. Despite the negligible difference in the speedup, ABC-optimizer performs much better in reducing L1 ICM and L2 ICM. Specifically, with about 30% more reduction for both L1 ICM and L2 ICM.

Xalancbmk One of the best improvements is achieved for this program. ABC-optimizer gives speedup of about 5% whereas CF-optimizer gives less than 2%. This is well-justified by the 47% more reduction in L1 ICM and about 20% more reduction in L2 ICM.

Python Thanks to the comprehensive set of reference inputs, ranging with respect to L1 ICM ratio, from 0.002% for *call_simple* to 15% for *mako*, we can have a more accurate study over ABC-optimizer. As we go from the inputs on the left to the inputs on the right, i.e. from less ICM ratios to larger ones, the speedup gained by ABC-optimizer more significantly outweighs the speedup gained by CF-optimizer.

From *call_simple* to *nbody*, CF-optimizer outperforms ABC-optimizer in average. From *unpack_sequence* to *bzr_startup*, ABC-optimizer slightly outperforms CF-optimizer, and finally, from *slowunpickle* to *mako* ABC-optimizer significantly performs better than CF-optimizer. Notably, among this tier, the best improvements are seen for *slowpickle* (8%), *mako* (11%), and *django* (12%). Having the best improvement for *django* is reasonable as the training input is *django* itself. Two exceptions in this tier are *normal_startup* and *bzr_startup*. However, even for these two inputs, in terms of L1 ICM and L2 ICM, the program is benefited more from ABC-optimizer than it is from CF-optimizer. Thus the reason for the poor performance of ABC-optimizer must have been something else.

The results for L1 ICM are even more striking. All but two of the inputs (*nbody* and *unpickle_list*) get more reduction from ABC-optimizer than they get from CF-optimizer, and ABC-optimizer increases L1 ICM for only one input(*nbody*).

The L2 ICM results indicate almost the same reduction for both of the optimizers with ABC-optimizer performing better in average. Two strange cases are *unpickle_list* and *pickle_list* for which, despite of the improvement in speedup, the L2 ICM increases by 207% and 65% respectively.

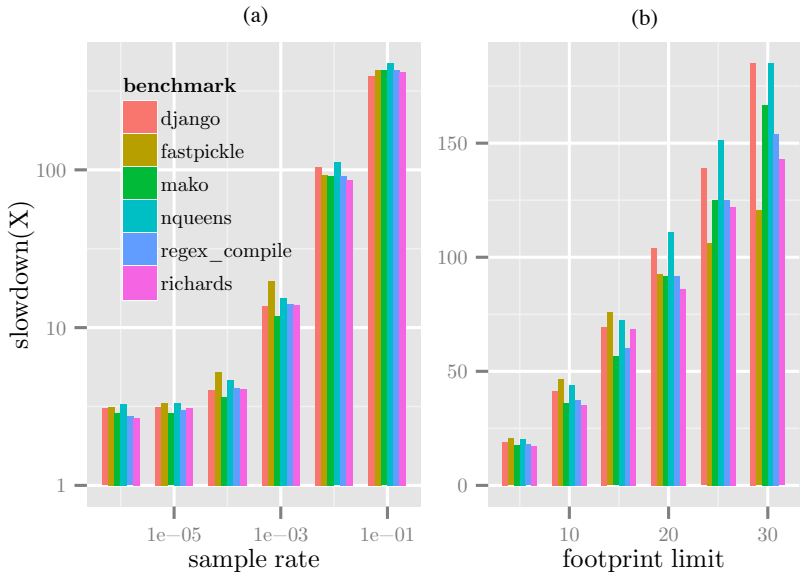

Figure 4: Normalized cost of profiling for sampling rates from 0.0001% to 10%, at footprint limit 20 (a), and for footprint limits from 5 to 30, at sample rate 1% (b)

Conclusions

- ABC-optimizer in general performs significantly better than CF-optimizer.
- As the instruction cache miss ratio increases, the performance of ABC-optimizer improves.
- The performance improvement is not expected to match proportionally with the miss reduction. Depending on how significant the instruction performance is in overall performance, a large miss reduction may result in a small speedup, or a small miss reduction may result in a large speedup.

4.4 Analysis Cost

Most of the optimization cost is that of the profiling analysis. Figure 4(a) shows the cost for sampling rates between 0.0001% to 10% in a log-log scale, for six test inputs of Python; *django*, *fastpickle*, *mako*, *nqueens*, *regex_compile*, and *richards*.

We observe that a portion of the slowdown, about 3x, is due to the base instrumentation overhead. Aside from this base overhead, the slowdown is linearly related to the sampling rate, confirming our analysis in Section 3.8.

Figure 4(b) shows the cost for footprint limits between 5 and 30, at sampling rate 1% for the same set of inputs. Recall from Section 3.8 that we established a quadratic relation between the footprint limit and the cost of the algorithm. However, we observe that the small sampling rate of 1% leads to subquadratic trends in slowdown, as expected. Among the inputs, the slowdowns for *django*, *mako*, and *nqueens* more closely follow a quadratic trend.

4.5 Sensitivity to Training Input

The Python benchmark does not come with a training input. Here we evaluate the effect of using different scripts for training. We choose the six scripts with highest L1 ICM ratio, *mako*, *nqueens*, *django*, *slowpickle*,

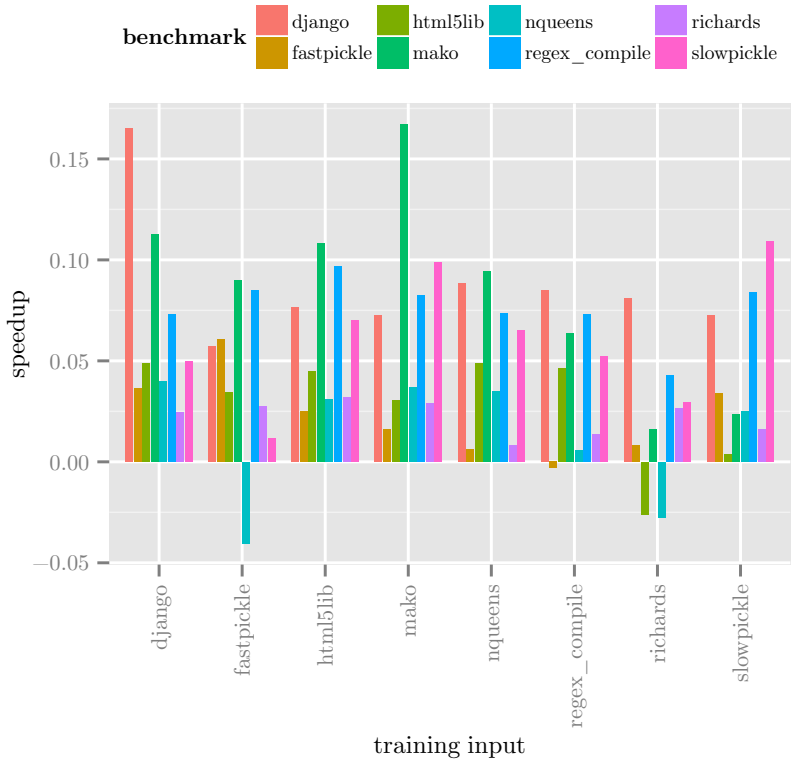

Figure 5: Cross training and optimization results. The Python interpreter is optimized based on training analysis of eight scripts, and each is tested on the same set of scripts.

html5lib, and *regex_compile*, along with two scripts with average L1 ICM ratio, *fastpickle* and *richards*. We measure the effect on the same set of scripts. Figure 5 shows the results.

We observe the best improvements when trained by *django*, *mako*, and *html5lib* where at least seven of the scripts speed up by at least 2.4% and the average speedup is more than 6%. On the next level, the training inputs *nqueens*, *slowpickle*, *fastpickle*, and *regex_compile* deliver an average speedup of between 4% and 5.3%. Finally, the worst training input is *richards* with average speedup of 1.9%. Four of the scripts gain improvement of 1.6% or less when trained by *richards* out of which two degrade by at least 2.5%.

Regardless of the training input, we see that the optimization improves performance in most cases and only occasionally degrade. This suggests that there are core functions in the interpreter that are used similarly by all scripts. As a result, we expect that the improvements are representative when using the optimized Python interpreter on other scripts.

However, we can also see a relation between a script's ICM and its training capability. *mako* and *django* turn out to be the best trainers while *richards* is the worst. This indicates that a larger footprint provide more useful information for optimization.

Another critical observation is that usually, the best improvement for one script is gained when the interpreter is trained with the same script (self-training). This is valid for *django*, *fastpickle*, *mako*, and *slowpickle*. For the rest of the scripts, the self-improvement is very close to the maximum (for all scripts but *regex_compile* the self-improvement is at most 0.5% below the maximum).

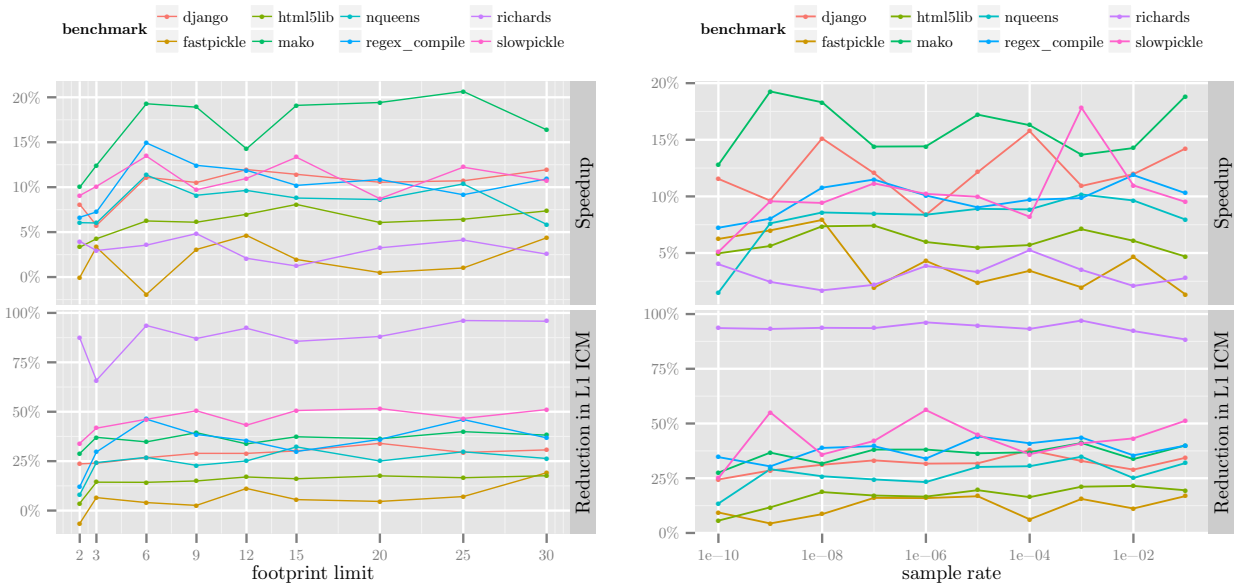

Figure 6: 8 programs optimized by self training affinity analysis for different parameters: (a) The footprint limit ranges between 2 and 30, and the sample rate is 1%. (b) The sampling rate ranges between 0.0001% and 10%, the footprint limit is 12.

4.6 Footprint Limit and Sampling Rate

The analysis has two parameters: the footprint limit and the sampling rate. Here we evaluate the sensitivity of performance to these two parameters. In this experiment, we use *self training*, where we train and test the ABC-optimizer on the same input. We use the Python interpreter and test the same 8 Unladen Swallow scripts that we used in Section 4.5.

Figure 6 shows the sensitivity in two graphs. The first uses the sampling rate 1% and varies the footprint limit between 2 and 30. The second limits the footprint to 12 and varies the sampling rate between 1E-10 and 0.01.

In Figure 6(a), as we go from the smallest footprint limit (2) to 6, we see consistent improvements for all the programs except *fastpickle* and *richards*. As we further increase the footprint limit, the speedup curves for *regex_compile* and *nqueens* slightly bend down, while for *django* and *html5lib* we see a slight increase in improvement up to footprint limit 15. For the rest of the programs the improvement does not change in either way and stays about the same value. We believe that this is caused by the coalescing of the affinity hierarchies. In the linearization process, we iterate to the largest footprint limit for each threshold. If the footprint limit is too large for a test program, the functions with high affinity in that footprint limit are probably too far to be grouped closely in the code layout.

We have also included the reductions in L1 ICM in Figure 6 for each parameterization. As we can see, up to footprint limit 6, the reduction in L1 ICM increases. This explains the consistent improvement in the speedup up to footprint limit 6. Moreover, for three of the programs, *mako fastpickle* and *richards*, the performance improvement curve very closely follow the L1 ICM reduction curve, though in different ways, due to their L1 ICM ratios. For *fastpickle* and *richards* a significant change in L1 ICM reduction triggers a smaller change in performance while for *mako* small changes in L1 ICM reduction trigger rapid change in performance.

Figure 6(b) shows the sensitivity to sampling rate. For the smallest sampling rate (1E-10), our training runs lead to sampling about 10 footprint windows. The results show that even with such a small number of sampled windows, we can get significant performance improvement. This also indicates that the python scripts have repetitive patterns of function calls.

The improvement increases for all but two of the programs when we increase the sampling rate from 1E-10 to 1E-9. Up to the sampling rate 1E-7, we can see increase in the improvements for *html5lib*, *nqueens*, *regex_compile*, and *slowpickle*. From this point, the improvements are mostly stable and do not change significantly. One special case is *fastpickle* for which the best improvements happen for smallest sampling rates.

In general, we can see stable improvements across different sampling rates. However, there is not always a consistent relation that a higher sampling rate leads to the same or better code layout. In our study, we have discovered a case where our algorithm produces a worse result at a higher sampling rate. Consider the trace "xyxyxy..xyz ... abzabzabz". Any sampling point that falls at x, y would have a sampled window of size 3 that contains x, y, z. At a higher sampling rate, we have more windows containing x, y, z than with a, b, z, which may not lead to the best affinity partition.

5 Related Work

Traditionally a compiler compiles one function at a time. A major problem is to reduce the cost of branches. Code layout techniques include replication to remove conditional branches (Mueller and Whalley, 1995) and branch reordering to facilitate branch prediction (Jiménez, 2005). A common tool is hot path profiling (Ball and Larus, 1994). These techniques aim to improve the speed at which a processor core fetches the next instruction. They may improve both code layout and code optimization. However, they do not maximize the utilization of instruction and data cache.

Zhang et al. reorganized the layout of basic blocks based on their reference affinity and showed improvements over superblock-based code layout (Zhang et al., 2006). For cache utilization, it is important to consider cold (infrequent) paths as well as hot paths. The size of the cold code is much greater, and most misses happen along cold paths. Reference affinity finds the pattern of co-occurrence and places related blocks into the same cache block. However, the previous technique is intra-procedural and does not optimize the global code layout. The previous results were measured in miss counts in simulation. There was no report of performance improvement.

Temporal-relation graph (TRG) is a model to improve the function layout across the entire program, although in transformation the original TRG technique is local since it adds space inside a function (Gloy and Smith, 1999). The goal of TRG is to minimize cache conflicts, not to maximize spatial locality.

As a problem, global code layout is similar to global data layout. A theoretical result shows that optimal placement is NP-hard and difficult to approximate (Petrank and Rawitz, 2002). Code layout optimization has the same complexity. Effective techniques have been developed for specific types of computations, including non-linear array layout (Frens and Wise, 1997, 2003; Chatterjee et al., 1999), cache-conscious layout for trees (Chilimbi et al., 1999a), cache-oblivious layout for B-trees (Bender et al., 2000), and space-filling curves for physics data (Jin and Mellor-Crummey, 2005; Mellor-Crummey et al., 2001). More general solutions include consecutive packing, graph partitioning and sparse tiling, reviewed and evaluated in (Strout et al., 2003; Han and Tseng, 2006).

Cache-conscious data placement has been developed for general purpose code, based on access frequency (Chilimbi et al., 1999b; Seidl and Zorn, 1998; Huang et al., 2004) and pairwise affinity (Calder et al., 1998; McIntosh et al., 2006). A metric called *neighbor affinity probability (NAP)* is defined to quantify program access to select whether to split an array of structures (Rabbah and Palem, 2003). Reference affinity is used to choose the best structure splitting, not just whether to split (Zhong et al., 2004). A more elaborate

model is used by ASLOP to consider affinity between both structure fields and structure instances (Yan et al., 2011).

Reference affinity is a hierarchical locality model (Zhong et al., 2004; Zhang et al., 2006). Previous work considered mostly strict reference affinity but also gave a sampling technique to find partial affinity based on a fixed threshold. We generalize the partial affinity to include all thresholds and give a new analysis algorithm that has the same asymptotic complexity as the previous analysis for a single threshold.

6 Summary

We have presented a new trace based algorithm for exploiting partial reference affinity. It efficiently analyzes the co-occurrence in all sampled windows. It combines the affinity in all footprint limits and all partial affinity thresholds. We have implemented the algorithm in a compiler and found it effective at improving the performance of the Python interpreter and xalancbmk and to a lesser extent, the Perl interpreter and GCC. The optimization performs significantly better than the call frequency based optimization, in general, and it does not cause significant slowdowns. It is robust across different tests, optimization parameters and cross-training relations.

References

Ball, T. and J. R. Larus. 1994. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4).

Bender, M. A., E. D. Demaine, and M. Farach-Colton. 2000. Cache-oblivious b-trees. In *Proceedings of Symposium on Foundations of Computer Science*.

Calder, Brad, Chandra Krintz, Simmi John, and Todd M. Austin. 1998. Cache-conscious data placement. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149.

Chatterjee, S., V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. 1999. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of International Conference on Supercomputing*.

Chilimbi, T. M., M. D. Hill, and J. R. Larus. 1999a. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. Atlanta, Georgia.

Chilimbi, Trishul M., Bob Davidson, and James R. Larus. 1999b. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24.

Cooper, Keith and Linda Torczon. 2010. *Engineering a Compiler, 2nd Edition*. Morgan Kaufmann.

Frens, Jeremy D. and David S. Wise. 1997. Auto-blocking matrix-multiplication or tracking BLAS3 performance with source code. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216.

Frens, Jeremy D. and David S. Wise. 2003. QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–154.

Gloy, Nicholas C. and Michael D. Smith. 1999. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027.

Han, Hwansoo and Chau-Wen Tseng. 2006. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618.

Huang, Xianglong, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: improving program locality. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 69–80. ACM Press, New York, NY, USA.

Jiménez, Daniel A. 2005. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 107–116.

Jin, Guohua and John M. Mellor-Crummey. 2005. SFCGen: A framework for efficient generation of multi-dimensional space-filling curves by recursion. *ACM Trans. Math. Softw.*, 31(1):120–148.

Lattner, Chris and Vikram S. Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 129–142.

McIntosh, Nathaniel, Sandya Mannarswamy, and Robert Hundt. 2006. Whole-program optimization of global variable layout. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 164–172. ACM, New York, NY, USA.

Mellor-Crummey, J., D. Whalley, and K. Kennedy. 2001. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming*, 29(3).

Mueller, Frank and David B. Whalley. 1995. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 56–66.

Petrank, E. and D. Rawitz. 2002. The hardness of cache conscious data placement. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Pettis, K. and R. C. Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Rabbah, R. M. and K. V. Palem. 2003. Data remapping for design space optimization of embedded memory systems. *ACM Transactions in Embedded Computing Systems*, 2(2).

Seidl, Matthew L. and Benjamin G. Zorn. 1998. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23.

Strout, M. M., L. Carter, and J. Ferrante. 2003. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257. San Diego, CA.

Yan, Jianian, Jiangzhou He, Wenguang Chen, Pen-Chung Yew, and Weimin Zheng. 2011. ASLOP: A field-access affinity-based structure data layout optimizer. *SCIENCE CHINA Info. Sci.*, 54(9):1769–1783.

Zhang, Chengliang, Chen Ding, Mitsunori Ogihara, Yutao Zhong, and Youfeng Wu. 2006. A hierarchical model of data locality. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–29.

Zhang, Chengliang, Yutao Zhong, Chen Ding, and Mitsunori Ogihara. 2004. Finding reference affinity groups in trace using sampling method. Technical report, Department of Computer Science, University of Rochester.

Zhong, Yutao, Maksim Orlovich, Xipeng Shen, and Chen Ding. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–266.