All Timescale Window Co-occurrence

Efficient Analysis and a Possible Use

(Position Paper)

Yumeng (Lucinda) Liu
University of Rochester
Rochester, New York 14627
yliu114@u.rochester.edu

Daniel Busaba
University of Rochester
Rochester, New York 14627
dbusaba2@u.rochester.edu

Chen Ding
University of Rochester
Rochester, New York 14627
cding@ur.rochester.edu

Daniel Gildea
University of Rochester
Rochester, New York 14627
dan@gildea.com

ABSTRACT

Trace analysis is a common problem in system optimization and data analytics. This paper presents new efficient algorithms for window co-occurrence analysis, which is to find how likely two events will occur together in time windows of different lengths. The new solution requires a linear time preprocessing step, after which, it only takes logarithmic space and constant time to compute co-occurrence of a data pair in windows of any given length. One potential use of the new analysis is to reduce the asymptotic cost in affinity-based memory layout.

CCS CONCEPTS

•**Theory of computation** → Caching and paging algorithms;

KEYWORDS

program analysis, affinity, memory layout

ACM Reference format:
Yumeng (Lucinda) Liu, Daniel Busaba, Chen Ding, and Daniel Gildea. 2018. All Timescale Window Co-occurrence. In *Proceedings of CASCON, Markham, Ontario, Canada, October, 2018 (CASCON'18)*, 5 pages.
DOI:

1 INTRODUCTION

A common problem in computer science is trace analysis, where a trace is a continuous sequence of events. An example is the sequence of memory requests made by a program during execution. Other examples include a sequence of functions executed by an application, a sequence of data requests to a web server, a series of objects displayed in a video stream, or a series of words in a document.

This paper presents new efficient algorithms for window co-occurrence analysis, which is to find how likely two events will occur together in a trace, i.e. the likelihood that two events will appear in the same time window. We can express this likelihood by a conditional probability: if a window contains event *a*, what is the probability that this window also contains event *b*?

In co-occurrence analysis, it is important to consider the timescale. If the timescale includes the whole trace, then any pair of events co-occur. On the other hand, if the timescale includes just one event, then there is no co-occurrence. The main strength of the new analysis is that it analyzes co-occurrence in *all* timescales, from the smallest to the largest.

The new analysis has the following steps:

- all-timescale solo occurrence
  - preprocessing phase: process the trace once and store the *reuse times*
  - analysis phase: calculate the result for different given window lengths, using the *reuse times*
- all-timescale co-occurrence
  - preprocessing phase: process the trace once and store the combined result of *switch times* and *inter-switch times.*
  - analysis phase: calculate the result for different given window lengths, using the combined times

One use of co-occurrence analysis is to analyze and optimize the memory layout of a program. Modern processor performance is dependent on cache performance which depends on cache block utilization. Co-occurrence analysis finds a relationship between data accesses in a trace called reference affinity. Then, program data can be placed so that the most related data are within the same cache block. Reference affinity has been used extensively in past work to improve cache performance. With IBM Z's block size being 256 bytes, quadruple Intel x86's 64-byte block size, we expect using an optimized memory layout to be significantly more beneficial to system performance. Furthermore, different levels of the same memory hierarchy has different block sizes, e.g. 4KiB pages at the virtual memory. All-timescale co-occurrence analysis can be used to improve memory layout for all block sizes.

2 PROBLEM STATEMENT

A program execution is represented by its memory access trace, which is a sequence of memory addresses. In general, we consider a trace of elements, and the elements may appear repeatedly in the trace. We define the following symbols:

n:	the length of the trace, i.e. the number of elements
m:	the number of distinct elements

The logical time for each element is defined as its index, starting from 1.

For the analysis, we measure the following

- solo occurrence probability $P(a \in W_x)$, which is the fraction of length-x windows that contain element a
- co-occurrence probability $P(\{a, b\} \subseteq W_x)$, or equivalently $P(a \in W_x \land b \in W_x)$, which is the fraction of length-x windows that contain a pair of elements a and b

W_x is the working set defined by Coffman Jr. and Denning [1], who defined $W(t, T)$ to be set of data used in the window of length T ending at t. W_x is the same as $W(t, x)$ for an unspecified t.

We call the parameter $x \geq 0$ the timescale. For example, $P(\{a, b\} \in W_x)$ is the frequency of joint appearance of a and b at timescale x. A conditional probability can be computed as follows:

$$P(b \in W_x | a \in W_x) = \frac{P(a \in W_x \land b \in W_x)}{P(a \in W_x)} \quad (1)$$

In any window of length x, if it contains a, $P(b \in W_x | a \in W_x)$ is the probability that the window also contains b.

This conditional probability may be useful in a lot of areas such as affinity-based memory layout. In order to calculate the conditional probability, the most intuitive way is to process the trace with the specified window length x, and count how many windows contain the element a and how many windows contain both elements. By using this method, we need to loop through the trace every time for a new x. For each loop, the time complexity is at least $O(n)$.

Our techniques only require one pass of a trace, reducing both time and space complexity. Our techniques are divided into two phases: the preprocessing phase and the analysis phase. The preprocessing phase is the one-pass processing of the trace to store necessary information. The analysis phase is using the information to calculate the solo occurrence and co-occurrence for all window lengths x.

3 COUNTING SOLO OCCURRENCE

Our goal is to process the trace only once and record the necessary information we need, and then we can calculate the conditional probabilities for any element with any window length just based on the information instead of analyzing the trace again.

The information we collect includes the first access time and the last access time of each element. In addition, we collect the cumulative reuse times of all elements. For each trace element, if it is not a first access, it has a *reuse time*, which is the time interval between the current time and the last time the element is accessed.

Before explaining the solution, we first show an example of window counting. Figure 1 shows an occurrence of an element a in a trace. We want to count the number of length-x *target windows* (which contain this occurrence a). The earliest target window is $w1$, which has a as its last element. The last target is $w2$, which has it as the first element. All sliding windows from $w1$ to $w2$ contain a. Their starting points are shown by the red line, which extends

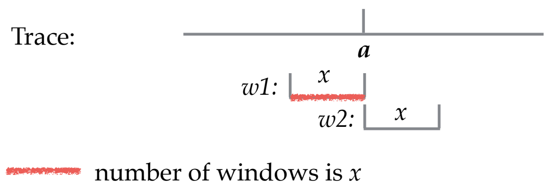

Figure 1: Counting the number of target windows which contain a specific occurrence of a. $w1$ is the earliest, and $w2$ the latest. The red line gives the count.

from the starting point of $w1$ to $w2$. In this illustration and later ones, the range of windows are shown by the first window on the first line and the last window on the second line, and the red line represents the count of target windows (and their starting points).

For a single occurrence, the number of target windows is actually just the window length x, which is the length of the red line in Figure 1.

Now, we consider the problem of window counting in a general situation that windows might contain different numbers of a. The general case is shown in Figure 2, where a is first accessed at time f_a, last accessed at time $n - l_a$, and reused a few times in between.

It is obvious that for window length x, we cannot simply use x as the number of windows that contain a. The solution is simpler if we solve the complement problem, i.e. counting windows that do not contain a. We call them *absence windows*, and the windows containing a are *target windows*. The number of target windows is the total number of windows minus the number of absence windows.

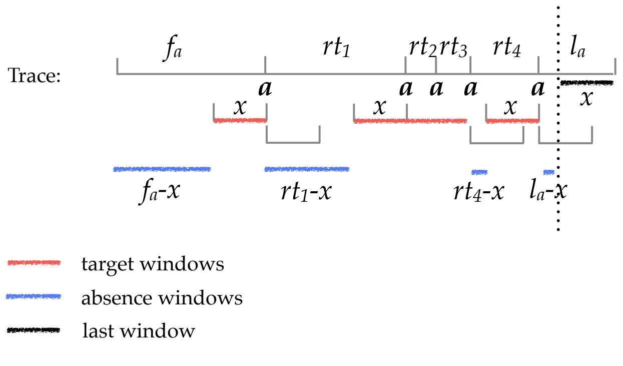

Figure 2: General cases of window counting for all accesses of a. The number of absence windows is computed by the first access time, the last access times or the reuse times minus x whenever it is greater than x.

In Figure 2, the red line denotes the number of target windows as we have explained previously. The blue line denotes the absence windows. Finally, the black line represents the last window for the whole trace, so the vertical dotted line means the starting point of the last window, which should be the ending point for counting.

The key insight is that each blue line can be computed by a formula of a common type, which is x subtracted from a time, where the time is either first/last-access time or the reuse time. The reason is that the absence windows (blue line) only happen if the time, either its reuse time rt, first access time f_a or last access time l_a, is larger than the window length x. For example, in Figure 2, rt_1 is larger than x and contains $rt_1 - x$ absence windows. rt_2 is less than x and does not contain any absence window.

Therefore, the number of absence windows is time $t - x$ if time $t > x$. The complete count of absence windows considers three types of time, given by the numerator in Eq. 2 as follows. The equation computes $P(a \notin W_x)$ as the absence-window count divided by the number of windows, which is $n - x + 1$. The solo occurrence probability $P(a \in W_x)$ is its complement, given by Eq. 3.

$$P(a \notin W_x) = \frac{\left(\sum_{i=x+1}^{n-1} (i-x)rt(i,a)\right) + (f_a - x)I(f_a > x) + (l_a - x)I(l_a > x)}{n - x + 1} \quad (2)$$

$$P(a \in W_x) = 1 - P(a \notin W_x) \quad (3)$$

The symbols are as follows:

- $rt(i,a)$: the number of reuse times of element a that equal to i.
- f_a: the first-access time of the element a (counting from 1).
- l_a: the *reverse* last-access time of the element a. If the last access is at position i, $l_a = n - i$, that is, the first-access time in the reverse trace (counting from 1).
- $I(p)$: the indicator function equals to 1 if p is true; otherwise 0.

4 COUNTING CO-OCCURRENCE

To measure co-occurrence, we follow the similar idea of solo occurrence counting. Instead of counting windows containing one element, we count windows containing a *switch*. A *switch* between a pair of elements, a and b, and is defined by the time interval that starts on one element and ends on the other, without any a or b in between. For example, the sequence $aabb$ has one switch (from the second a to the first b), and aba has two switches. We have a full solution for counting co-occurrence but choose to not include it in this position paper.

For each switch, the *switch time st* is the length of the interval, which is the position index of second element minus the position index of first element.

The switch time facilitates co-occurrence counting. First, any window containing both a, b must contain a switch. Hence counting co-occurrence means counting switch occurrence. Second, no window can contain a data pair of a switch when the switch time is larger than or equal to the window length x. Once we collect switch times, we will count co-occurrence windows by considering only switches whose switch time is smaller than window length x.

Figure 3 shows an example of co-occurrence counting. It shows two switches of a, b, and the target windows of the second switch, with $w1$ being the earliest target window and $w2$ the latest. The number of enclosing windows is $x - st$, where x is the window length, and st the switch time. Notice here we don't treat the time interval from first b to second a as a switch since its switch time st is larger than window length x.

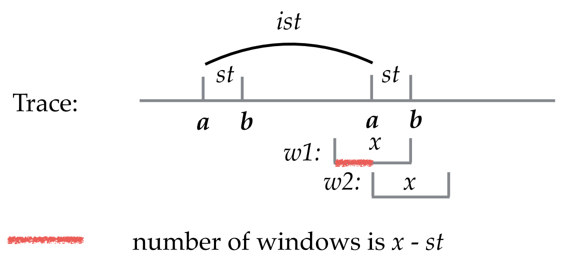

Figure 3: The number of windows containing a switch of data pair a and b is $x - st$, where x is the window length and st the switch time. ist is the inter-switch time between the two switches, which is the time from the first element of first switch to the first element of second switch.

Solo occurrence can now be seen as a special case of co-occurrence, where the switch is reduced to a single element and $st = 0$. In this special case, the number of windows containing the very next element is also $x - st$ (as in Figure 3), where st is always 0.

Now, we can treat a co-occurrence of a and b as a solo occurrence of a switch. We use a similar strategy as before. Recall that for solo occurrence, we counted absence windows using reuse time minus window length x for any reuse time larger than x.

For similar reason, we need the reuse time in solo occurrence counting, we need the time between consecutive switches in co-occurrence counting. We define the *inter-switch time ist* as the time difference between the *first* elements of two consecutive switches.

In solo occurrence counting, the number of absence windows for each reuse time rt is $rt - x$, if $rt > x$. In co-occurrence counting, the number of absence windows has three cases. In Case 1, $x > st$ and $ist > (x - st)$, the count is $ist - (x - st)$, as seen in Figure 3. The number of absence windows is the number of windows minus the number of target windows, and the number of target windows is $x - st$ as explained previously. In Case 2, $x > st$ and $ist \leq (x - st)$, the count is 0. Finally, in Case 3, $x \leq st$, the count is ist. The following table shows the three cases:

	$ist > (x - st)$	$ist \leq (x - st)$
$x > st$	$ist - (x - st)$ (Case 1)	0 (Case 2)
$x \leq st$	ist (Case 3)	not possible

5 STORING TIME

The asymptotic cost of co-occurrence analysis depends on how its inputs are stored. These inputs are distributions of time, including the reuse time and the first- and last-access times in in Eq. 2. Similar time representations are used to compute co-occurrence.

We store a distribution of time in a histogram. We define the *size* of a histogram by the number of buckets. The maximal value of time is the length of the trace n. A fully precise histogram may have n buckets. We can reduce the size of a histogram to either bounded or logarithmic of the maximum value by approximation.

A basic solution divides the full value range evenly. This solution is constant size and general. It may waste space when values are sparsely distributed. A specialized solution is a *reference histogram*, which sorts all reuse distances by their values and divides them evenly into 1000 bins, so each bin stores exactly 0.1% of reuse distances [11]. A reference histogram may still waste space because two adjacent bins may store identical values. Another solution is recursive division, which stops dividing a group when its values are identical [4].

Logarithmic size histograms are commonly used. In the basic solution, the ith bin stores the range $[2^i, 2^{i+1}-1]$. There are at least two ways to improve precision. The first is to record the average value in each bin and assume a constant or linear distribution by the values in the range (fitted to give the same average) [2]. The second is a *k-sublog* histogram, which further divides a power-of-two range into 2^k sub-ranges for a pre-determined constant $k > 0$ [6, 7]. For example, a 8-sublog histogram uses 256 sub-ranges and is accurate from 0 to 511 and then divides each successive power-of-two ranges into 256 bins. Note that the size of sublog bins still increases exponentially, but a user can trade off between histogram size and precision by adjusting k, and the asymptotic space cost is always logarithmic of the maximum value in the histogram.

6 COMPLEXITY ANALYSIS

We now analyze the time and space requirements for single date items.

For single data items, preprocessing goes through an execution trace once to measure the reuse time distribution $rt(i, a)$ for each data a and its first- and last-access time. The time complexity is $O(n)$. It needs a hash table to measure the reuse time (by storing the last access time of each data item), so the space complexity is $O(m)$ for the hash table. As explained in Section 5, a time distribution can be stored in a histogram of size $O(\log n)$, so the result of the preprocessing takes $O(m \log n)$ space, one histogram for each data item.

For the analysis phase, the sum of Eq. 2 is replaced with a sum over buckets in the historgram. Therefore, the time complexity is $O(\log n)$, but can be reduced to $O(1)$ by storing cumulative counts representing the first k terms of Eq. 2.

For co-occurrence of pairs of data times, the analysis is similar, which we do not include in the position paper.

7 AFFINITY-BASED MEMORY LAYOUT

Almost all modern computers use cache blocks of at least 64 bytes, making the utilization an important problem. If only one word is useful in each cache block, a cache miss will not serve as a prefetch for other useful data. Furthermore, the program would waste up to 93% of memory transfer bandwidth and 93% of cache space, causing even more memory access. The problem is worse on IBM Z series, whose block size is 256 bytes, quadruple the 64-byte block size used by other systems.

To improve cache utilization we need to group related data into the same cache block. A common technique is affinity analysis. *Reference affinity* measures how close a group of data are accessed together in an execution.

Zhong et al. [10] defined k-distance analysis and proved that it gives a unique partition of program data for each distance k. When the value of k decreases, the reference affinity gives a hierarchical decomposition and finds data sub-groups with closer affinity. The requirement is *strict* in that for a group of data elements to have reference affinity, they *all* need to *always* be accessed closely together.

Zhang et al. [8] relaxed the criterion and defined weak reference affinity, which occurs when at least a fraction of a group of data are used together. The fraction is specified by a parameter. Every fraction parameter gives a separate hierarchical partition of data elements. When this parameter equals one, it gives the strict reference affinity.

Affinity analysis has been adapted in the IBM compiler to improve the array layout [5] and shown effective for improving the instruction layout [9].

Both the strict and the weak reference affinity are computationally hard problems, i.e., finding the affinity groups is NP-hard [9]. Previous work has simplified the problem by approximation, e.g. checking a sufficient but not necessary condition in k-distance analysis [10], and by restricting analysis only to arrays [5]. Full affinity analysis is prohibitively expensive for a large data set.

Recently, Lavaee [3] developed a solution for large-scale affinity analysis. It uses one-pass profiling to measure pairwise affinity at different levels. The pairwise affinity is defined by window co-occurrence probability. The definition uses *affinity windows.* First, the *window WS size* is the working-set size of a window, i.e., the number of unique memory objects accessed in the window. The affinity window $AW(t, \ell)$ is the *shortest* window that starts from t and whose WS size is ℓ. The set of all such windows is the *affinity window set* $AWS(t, \ell)$.

The conditional probability $P(b \in W_x | a \in W_x)$ of Eq. 1 provides an alternative definition of pairwise affinity. Each time window is an affinity window. In $AW(t, \ell)$, ℓ is simply the window length. The set $AWS(t, \ell)$ includes all windows of length ℓ. To distinguish, we say that Lavaee affinity uses WS windows, and Eq. 1 uses time windows.

Now we can state an important benefit of this work. It provides till now the highest asymptotic efficiency for affinity analysis. Assuming we analyze the pairwise affinity for a constant number of data pairs, the complexity of this work and the earlier studies are as follows.

- *Time-window affinity.* The analysis takes $O(n)$, where n is the length of the trace. As this paper shows, the analysis computes the conditional probability for all timescales.
- *WS-window affinity.* The analysis takes $O(nS^2)$, where S is the maximal affinity level [3].
- *Strict and weak reference affinity.* The analysis is NP-hard [9].

Time-window affinity does not count distinct data elements, i.e. data size, in a window, which is the reason for its efficiency over WS-window affinity. However, for cache performance, data size matters. One possible remedy is for time-window affinity to estimate the data size using the window length.

8 SUMMARY AND FUTURE WORK

Trace analysis is a common problem in computer science, and window co-occurrence analysis for all timescales is a new addition. In this paper, we used conditional probability to represent the likelihood that two events will occur in the same time window. To calculate the conditional probability, we show how to measure solo occurrence of individual data items and co-occurrence of data pairs.

We use the reuse times of the elements to calculate the solo occurrences for different timescales x, and we use similar time representations to count the co-occurrences. For both calculations, we only need to process the trace once to store a set of time distributions.

By storing time distributions in histograms, we can reduce the space complexity to logarithmic of the trace length and time complexity to constant per timescale, after a linear-time preprocessing step.

The further work of this paper may be trying to find algorithms that help define relationship between more than two elements in a trace, i.e. co-occurrences of a group of elements, where the group size is larger than two. We plan also to study the uses of co-occurrence information in memory layout optimization as discussed in Section 7 and find additional uses.

ACKNOWLEDGEMENT

The authors wish to thank Sreepathi Pai, Michael Scott, Yuhao Zhu, Wentao Cai and the anonymous reviewers of CASCON 2018 for the helpful comments and suggestions. The work is supported in part by the National Science Foundation (Contract No. CCF-1717877, CCF-1629376) and an IBM CAS Faculty Fellowship.

REFERENCES

[1] E. G. Coffman Jr. and P. J. Denning. *Operating Systems Theory.* Prentice-Hall, 1973.

[2] C. Fang, S. Carr, S. Önder, and Z. Wang. Instruction based memory distance analysis and its application. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 27–37, 2005.

[3] R. Lavaee. *Profile-Guided Memory Layout: Theory and Practice.* PhD thesis, Computer Science Dept., Univ. of Rochester, January 2018.

[4] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, 2004.

[5] X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight reference affinity analysis. In *Proceedings of the International Conference on Supercomputing*, pages 131–140, 2005.

[6] X. Xiang, B. Bao, T. Bai, C. Ding, and T. M. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 91–102, 2011.

[7] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 343–356, 2013.

[8] C. Zhang, Y. Zhong, C. Ding, and M. Ogihara. Finding reference affinity groups in trace using sampling method. Technical Report TR 842, Department of Computer Science, University of Rochester, 2004. *presented at the 3rd Workshop on Mining Temporal and Sequential Data, in conjunction with ACM SIGKDD 2004.*

[9] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu. A hierarchical model of data locality. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–29, 2006.

[10] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. 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, 2004.

[11] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 31(6):1–39, Aug. 2009.