A Hierarchical Model of Data Locality

Chengliang Zhang, Yutao Zhong, Mitsunori Ogihara and Chen Ding

Computer Science Department, University of Rochester
{zhangchl,ytzhong,ogihara,cding}@cs.rochester.edu

Abstract

To study data placement on memory hierarchy, we present a model called *reference affinity*. Given a program trace, the model divides program data into hierarchical partitions (called affinity groups) based on a parameter k, which specifies the number of distinct data elements between accesses to members of each affinity group. Trivial solutions exist for the two ends of the hierarchy. At the top, when k is no less than the data size, all program data belong to one affinity group. At the bottom, when k is 0, each element is an affinity group.

We present two theoretical results. The first is the complexity. We show that finding and checking affinity groups are in P when $k = 1$ and $k = 2$. When $k = 3$, the checking problem is NP-complete, and the finding problem is NP-hard. The second is the uses. We show that reference affinity captures the hierarchical data locality from the trace of a hierarchical computation. As additional evidence, we cite empirical results for general-purpose programs.

1 Introduction

While program data are defined in a uniform address space, they are stored in a memory hierarchy that often includes registers, on-chip and off-chip cache, virtual memory, disk, and network buffers. Different memory levels come with different sizes and configurations. A programmer may not know the exact parameters or even the existence of some hardware cache. Furthermore, a mobile program can migrate in a distributed system and hence needs to run well on machines with different memory configurations. As programming for specific memory hierarchies becomes increasingly untenable, it arises the need for *hierarchical data locality*, where the program data are recursively decomposed into smaller groups based on the inherent locality in computation. By blocking data at all levels, the hierarchical locality subsumes the layout schemes designed for specific memory hierarchies.

The past locality models are not general or not hierarchical. The general models are based on the frequency of data access for data elements (as early as [10]), pairs (for example in [8]), or streams [5]. The frequency models use pre-determined thresholds and are not naturally hierarchical. Hierarchical data placement has been successfully used but only for specific problems including matrix multiplication, factorization, wavelet transform [4], N-body simulation [13], and search trees [2]. In the last work, Bender et al. coined the term *cache-oblivious data layout*. In this paper, we study a general and hierarchical model that is based on program traces.

The locality model is called *reference affinity*, which measures how close a group of data is accessed together in a reference trace. The closeness is parameterized by k, which is the volume distance between adjacent accesses to group members. The *volume distance* between two points on a trace is defined as the amount of data accessed from the first point to the second. Changing k, reference affinity gives a *hierarchical* partition of program data. When k is equal to or greater than the data size, all data belong to one affinity group. When k is 0, each data element is an affinity group.

We present two theoretical results. First, we characterize the complexity of reference affinity, in particular, the problems of linking two data elements, and checking and finding affinity groups. We give polynomial-time algorithms for cases $k = 1$ and $k = 2$. We prove that the problems are either NPC or NP-hard when $k \geq 3$. Second, we show that reference affinity automatically captures the hierarchical locality in divide-and-conquer algorithms. In addition, we cite empirical evidence that good approximation analysis exists for reference affinity in general programs.

The concept of volume distance was first defined by Mattson et al. [12] (named LRU stack distance) in 1970. It forms the foundation for virtual memory management [6] and cache design [18], because it efficiently measures the performance of fully associative cache of all sizes that uses the least-recent-used (LRU) replacement policy. Sleator and Tarjan [16] and Aggarwal et al. [1] showed that LRU policy is within a constant factor of the optimal cache management. An increasing number of studies in the past four years show that the volume distance allows accurate modeling of the memory behavior of complex programs (for example [3, 7, 11]).

The volume distance has been difficult for theoretical analysis because data may appear in different orders with different frequencies while still yielding the same volume distance. It raises interesting problems different from those in traditional graph and streaming domains. In this work, we solve these problems and make the volume distance the basic link between the patterns in computation and the locality in data.

The organization of computation, the modeling of memory hierarchy, and data layout schemes for specific problems have been extensively studied by the algorithm community [20]. Our model is limited to finding the data locality, but it is general because it is defined on program traces. Given a computation trace, our model helps a programmer to select and fine-tune the data layout or to guide a compiler or a virtual machine in automatic data placement. Many techniques exist for static, profiling-based, or dynamic exploitation of data locality. In this work, we study in theory how much we can characterize the inherent data locality.

2 Reference Affinity Model

An *address trace* or *reference string* is a sequence of accesses to a set of data elements. If we assign a logical time to each access, the address trace is a vector indexed by the logical time. We use letters such as x, y, z to represent data elements, subscripted symbols such as a_x, a'_x to represent accesses to a particular data element x, and the array index $T[a_x]$ to represent the logical time of the access a_x on a trace T. In the latter part of this paper, we use sequence and trace interchangeably.

Definition 1 Volume distance. *The volume distance between two accesses, a_x and a_y ($T[a_x] < T[a_y]$), in a trace T is the number of distinct data elements accessed in times $T[a_x], T[a_x] + 1, \ldots, T[a_y] - 1$. We write it as $dis(a_x, a_y)$. If $T[a_x] > T[a_y]$, we define $dis(a_x, a_y) = dis(a_y, a_x)$. If $T[a_x] = T[a_y]$, we define $dis(a_x, a_y) = 0$.*

For example, the volume distance between the accesses to a and c in the trace *abbbc* is 2. The volume distance is Euclidean. Given any three accesses in the time order, a_x, a_y, and a_z, we have $dis(a_x, a_z) \leq dis(a_x, a_y) + dis(a_y, a_z)$, because the cardinality of the union of two sets is no greater than the sum of the cardinality of each set. Next we define the condition that a group of data elements are accessed together.

Definition 2 Linked path. *A linked path in a trace is parameterized by a distance bound k. There is a linked path from a_x to a_y ($x \neq y$) if and only if there exist t accesses, $a_{x_1}, a_{x_2}, \ldots, a_{x_t}$, such that (1) $dis(a_x, a_{x_1}) \leq k \wedge dis(a_{x_1}, a_{x_2}) \leq k \wedge \ldots \wedge dis(a_{x_t}, a_y) \leq k$ and (2) $x_1, x_2, \ldots, x_t$, x and y are different (pairwise distinct) data elements.*

In other words, a linked path is a sequence of accesses to different data elements, and each link (between two consecutive members of the sequence) has a volume distance no greater than k. We call k the *link length*. We will later restrict $x_1, x_2, \ldots, x_t$ to be members of some set S. If so, we say that there is a k-linked path from a_x to a_y with respect to set S.

Sequence (1) shows an example sequence. Each $''\ldots''$ section represents a sequence of data elements other than a, b, and c. While a, b, and c are always accessed together in the trace, they are accessed in different sequences and frequencies, and their accesses are intermixed with other data accesses. Still, in each occurrence, there is a linked path connecting three elements with a link length 3. Next, we define them as a group that we call a *reference affinity group*.

$$\ldots abebc \ldots bffaac \ldots ccccbga \ldots \tag{1}$$

Definition 3 Reference affinity. *Given an address trace, a set G of data elements is a reference affinity group (i.e. they have the reference affinity) with the link length k if and only if*

1. *for any $x \in G$, all its accesses a_x must have a linked path from a_x to some a_y for each other member $y \in G$, that is, there exist different elements $x_1, x_2, \ldots, x_t \in G$ such that $dis(a_x, a_{x_1}) \leq k \wedge dis(a_{x_1}, a_{x_2}) \leq k \wedge \ldots \wedge dis(a_{x_t}, a_y) \leq k$*
2. *adding any other element to G will make Condition (1) impossible to hold*

Reference affinity groups give a unique and hierarchical partition of data, as proved by Zhong et al. in the form of the following three properties [21].

1. **Uniqueness** Given an address trace and a fixed link length k, the affinity groups form a unique partition of program data.
2. **Hierarchical structure** Given an address trace and two distances k and k' ($k < k'$), the affinity groups at k form a finer partition of the affinity groups at k'.
3. **Bounded access range** Given an address trace with an affinity group G at the link length k, any time an element x of G is accessed at a_x, there exists a section of the trace that includes a_x and at least one access to all other members of G. The volume distance between the two sides of the section is no greater than $2k|G| + 1$, where $|G|$ is the number of elements in the affinity group.

Having the definition of reference affinity, the problems of checking and finding reference affinity groups can be formulated as the following respectively:

Definition 4 Checking reference affinity groups *Given an address trace and reuse distance k, check if a given group of data elements belongs to the same reference affinity group with link length k.*

Definition 5 Finding reference affinity groups *Given an address trace and reuse distance k, find the reference affinity groups with link length k.*

A related decision problem with checking reference affinity groups is to test if two accesses is k-linked with each other:

Definition 6 *Given an address trace, a reuse distance $k \geq 0$ and two data accesses a_x and a_y, the Point-wise k-Linked Affinity Problem (*Pw-k-Aff*, for short) is the problem of testing whether a_x and a_y are k-linked in the trace.*

3 Hardness of Finding and Checking Reference Affinity Groups

The following theorems give the complexity of the linking, checking, and finding problems for different k. We include the basic ideas of the proofs and leave the full version in the appendix.

Theorem 1 *For each $k \geq 3$,* Pw-k-Aff *is* NP-*complete.*

We prove it by making a polynomial-time many-one reduction from a variant of 3-SAT problem, where every variable appears at most three times (an NP-complete problem) to our linking problem. The proof constructs a three-part reference trace. The first part forces a linked path to go through a set of elements we call "separators", whose later access cannot be used as links in the next two parts. The second part of the trace prepares a set of element triples, where they can model the truth values of 3-SAT expressions. It ensures that two data representing the opposite values of a truth variable cannot both be included in any possible linked path. The third part of the sequence models a given 3-SAT expression. A linked path exists if and only if there is a truth value assignment to satisfy the expression. The basic ideas behind the proof are to use logical variables to represent accesses rather than data and to use specially designed traces to enforce the logical consistency. The full proof is two-page long and given in the appendix. From Theorem 1, we can easily prove two corollaries.

Corollary 1 *For $k \geq 3$, the problem of checking reference affinity groups is* NP-*complete.*

Corollary 2 *For $k \geq 3$, the problem of finding reference affinity groups is* NP-*hard.*

Theorem 2 Pw-*2*-Aff *is* NL-*complete.*

Using the same polynomial-time reduction from Theorem 1, we can show that *2*-CNF-SAT can be reduced to Pw-*2*-Aff. The exact proof is in the appendix. This theorem shows that a polynomial algorithm exists for Pw-*2*-Aff. Then we have the following result, proved by the algorithm that follows.

Theorem 3 *For $k = 2$, the problem of finding reference affinity groups is in P.*

Algorithm 1 Finding reference affinity groups when k=2
procedure *FindReferenceAffinityGroup_2(T)*

```
1: {T is the trace, reuse distance k = 2}
2: initialize, no group is identified
3: while there exist ungrouped elements do
4:    put all such elements into a set G and pick one x randomly from this set;
5:    repeat
6:       if there is an element z not 2-linked to x with respect to G then
7:          remove z from G;
8:       else
9:          if there exist two elements y, z ∈ G such that an access of y is not 2-linked to any access
            of z with respect to G then
10:            remove z from group G.
11:         end if
12:      end if
13:   until G is unchanged
14:   output reference affinity group G.
```

15: ***end while***

endFindReferenceAffinityGroup_2

Algorithm 1 is polynomial-time. From Theorem 2, the linking problem, that is, testing whether a 2-linked path exists between two data accesses, can be solved in polynomial time. This algorithm needs a polynomial number of such tests. The algorithm gives correct reference affinity groups. First, it is easy to see that the groups found by this algorithm satisfy the first condition of reference affinity. To show every group is the largest possible, we show that the algorithm removes z correctly, so that G still includes only the reference affinity group that x belongs to. Removing z at step 7 is straightforward. The correctness of the removal of z at step 10 can be proved by contradiction. Suppose z belongs to the same group as x and should not be removed, we can construct a 2-linked path from every access of y to an access of z. This contradicts with the test at line 9. The detail of the proof is given in the appendix. From Theorem 3, we can get the following corollary.

Corollary 3 *For $k = 2$, the problem of checking reference affinity groups is in P.*

The complexity for $k = 1$ is as follows.

Theorem 4 Pw-*1*-Aff *can be solved in linear time.*

Theorem 5 *For $k = 1$, there is a polynomial-time solution for finding reference affinity groups.*

Here we give a naive method. Since $k = 1$, all of the groups appear in the sequence continuously, and two groups do not overlap. We sort the data elements according to their order of appearance in the trace. Then for every t (from the number of data elements to 1) consecutive data elements starting from the first data element, we check if it is a reference affinity group. Similarly, we find other affinity groups. The algorithm is given in the appendix. Finally, from Theorem 5, we have

Corollary 4 *For $k = 1$, the problem of checking reference affinity groups can be solved in polynomial time.*

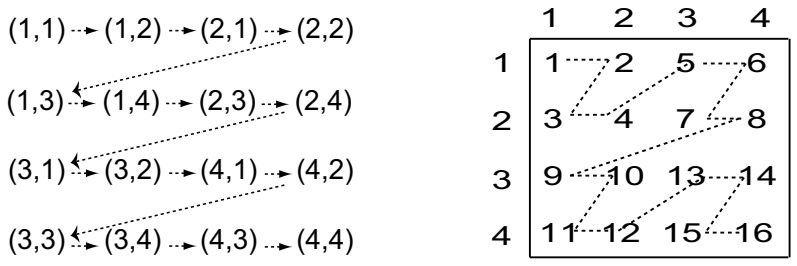
4 Use in Divide-and-Conquer Programs

The divide-and-conquer type of computations we consider are blocked and recursive algorithms for dense matrix operations, N-body and mesh simulation, and wavelet transform. The general form is given in Figure 1. The procedure takes a set of data such as matrices. It then divides the input data into smaller blocks and processes all or subsets of their combinations. For each subset, if the blocks are still large, it makes a recursive call to itself. The computation is hierarchical, so is its locality.

We show that reference affinity can reconstruct the hierarchical data locality from an execution trace, if the following two requirements are met by the hierarchical computation. First, once a block of $Data_i$ is accessed, all its sub-blocks are accessed before moving to the next block of $Data_i$. Second, the access order of sub-blocks is the same for the same block. For example, consider the multiplication of two matrices A and B. The computation, if starting from the left-matrix of A, must access all elements of the left sub-matrix of A at least once before accessing any element from the right sub-matrix of A. Still, it is free to access B or other data at the same time. The traversal order within A is the same, for example, Morton order. The traversal order in B can be different, for example, Hilbert order. In fact, non-nesting blocks in the same matrix have this freedom—the left sub-matrix can use a different order than the right sub-matrix.

```
Compute(D_1, D_2, ..., D_n) begin
  if the input data is above a threshold size
    divide D_1, D_2, ..., D_n into sub-blocks
    for some set of sub-block combinations
      Compute(subblock_i(D_1), subblock_j(D_2), ..., subblock_k(D_n))
    end for
  end if
end
```

Figure 1: The general form of the divide-and-conquer algorithm



(a) (b)

Figure 2: Trace of N-body simulation when $n = 4$

We use N-body simulation as an example, which calculates how particles interact and move in space. Most computation is spent on computing the direct interaction between each particle and its neighbors within a specific radius. The typical implementation divides the space into basic units. For ease of presentation, we assume each unit contains the same number of particles, and the program computes the interaction between all unit pairs. Our main result, Theorem 7, holds when units contain a different number of particles, and when interactions are limited within a radius.

In the following analysis, we assume a one-dimensional space. Higher dimensions can be linearized by using a space-fitting curve [13]. For simplicity, we assume that the space has $n = 2^t$ units, where integer t is non-negative. The N-body simulation trace is then of size 2^{2t+1}. As an example, we give the trace when $n = 4$ in Figure 2. The trace follows the Morton space filling curve, which is shown at the right part of Figure 2.

We define the division of units into sections. We call the set of units $i * m + 1$ to $i * m + m$ an m-section of data, where m is a power of 2 and i is a non-negative integer.

The N-body simulation trace can be expressed as a matrix, as shown in Figure 2 (b). The rows and columns are units, and the Morton space filling curve corresponds to the execution order. The interaction between a m-section and another m-section is computed at their product in the graph, which is a block of size m by m. We call it an m-block of computation. In a divide-and-conquer computation, each m-block contains a contiguous sequence of the computation trace.

Now we prove that reference affinity gives the hierarchical locality in divide-and-conquer algorithms, again using N-body simulation with n units as an example. The main theorem of the section will show the exact structure of the reference affinity hierarchy. As a shorter exercise, we first show that the reference-affinity hierarchy has more than a constant number of levels when n can be arbitrarily large.

Theorem 6 *For one-dimensional N-body simulation in the Morton order, the reference affinity has more than a constant number of levels when n is arbitrarily large.*

Proof Given a k that is a power of 2, we show that (a) every $\frac{k}{2}$-section of data belongs to a k-affinity group but (b) some m-section of data does not all belong to a k-affinity group. We prove Part (a) first. Every use of a $\frac{k}{2}$-section data is contained in a $\frac{k}{2}$-block of computation, which contains k distinct data. It is obvious that a k-linked path exists from any access to any other access in the $\frac{k}{2}$-block, therefore a $\frac{k}{2}$-section belongs to a k-affinity group.

We prove Part(b) by contradiction. Suppose for any m, a m-section of data belongs to a k-affinity group. We denote the first and the last data elements of the m-section as d_1 and d_m. According to the definition of reference affinity, there must be a k-linked path from the first access of d_1 to some access of d_m, the path has at most $m-1$ links, and the volume distance of each link is no more than k. The trace from the first access of d_1 to the first access of d_m includes at least a $\frac{m}{2}$-block, which has a length $\frac{m^2}{4}$. Hence the path of $m-1$ links spans at least $\frac{m^2}{4}$ data accesses, and there must exist a link that spans at least $\frac{m}{4}$ accesses. However, when m is large enough, it is impossible to bound the number of distinct data in $\frac{m}{4}$ contiguous accesses on the trace. The volume distance of the link must be greater than k. A contradiction. Therefore, the reference-affinity hierarchy has more than a constant number of levels when n can be arbitrarily large. ∎

Next we prove the exact structure of the reference affinity hierarchy. First, we give a key lemma needed by the final theorem. We call it the *insertion lemma*. It shows that the insertion of a new data access converts a link of length k into two shorter links.

Lemma 1 *Given two different data elements u and v; their accesses a_u and a_v where the volume distance from a_u to a_v is exactly k; and a third access a_x, which happens between a_u and a_v in the trace; then there exists an access a'_x between a_u and a_v such that the volume distance from a_u to a'_x is less than k, and the volume distance from a'_x to a_v is less than k.*

The insertion lemma states that a link from a_u to a_v of length k can be divided into two shorter links. In particular, for any data element x accessed along the path, there exists an access, a_x, such that it breaks the link into two shorter links. Not all accesses to x can be the breaking point. The proof considers all possible configurations of u, v, x, a_u, a_v and shows the placement of a_x in each case. The proof is quite long and mechanical, and a sketch is included in the appendix.

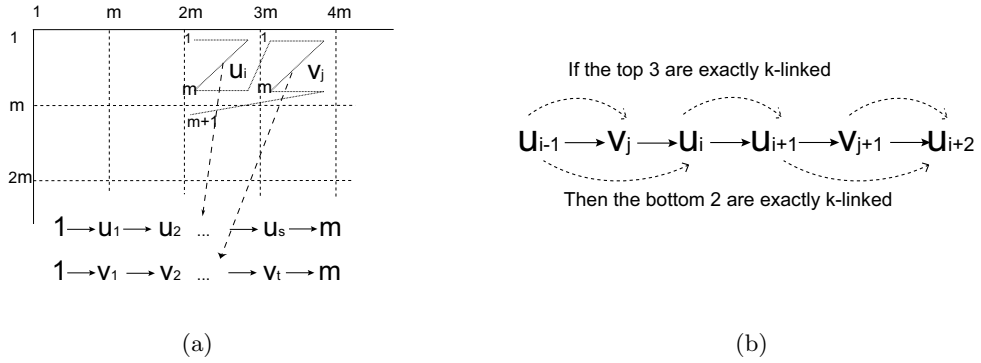
Next, the final theorem gives the exact structure of the reference-affinity hierarchy. It is the most important theoretical result, finishing the link between the linear, flexible concept of linked paths in a computation trace and the hierarchical, complex structure of locality in space.

Theorem 7 *Given N-body simulation of 2^s particles implemented using the divide-and-conquer technique or a space-fitting curve, the reference affinity hierarchy contains $s+1$ levels, where each 2^i-section belongs to a i-level affinity group.*

The proof is straightforward after proving the following lemma.

Lemma 2 *For any m-section, there exists a k, such that the m-section is a k-affinity group, but the 2m-section does not all belong to the k-affinity group.*

Proof Let k be the smallest reuse distance such that the m-section belongs to an affinity group. Without loss of generality, we assume the m-section and $2m$-section are the first such sections in the data space, as shown in Figure 3(a). Suppose the $2m$-section also belongs to the k-affinity



(a) (b)

Figure 3: Illustrations for proofs of Lemma 1 (the insertion lemma) and Theorem 7

group, we derive a contradiction by showing that the m-section belongs to a $k-1$-affinity group. It suffices to show that there is a $k-1$-linked path from the first access of 1 to the first access of m.

Because the $2m$-section are in a k-affinity group, there is a k-linked path from the first access of element 1 to some access of element $m+1$, as shown in Figure 3(a). The path is linked by at most one access of elements 2 to m. Now consider the two m-blocks of computation in the figure marked with u_i and v_j. They divide the path into two parts. By adding an ending point at the first access of m in the u_i block, and a starting point at the first access of 1 in the v_j block, we cut the k-linked path from 1 to $m+1$ into two k-linked paths from an access of 1 to an access of m. The two paths are shown at the bottom of Figure 3(a). The intermediate links in the two paths are $u_1, \ldots, u_s$ and $v_1, \ldots, v_t$. We map the v_i path in the v_j block to the u_i block. We now have two k-linked paths from the first access of 1 to the first accesses of m. The links are accesses to different data elements.

We construct a $k-1$ linked path from the first access of 1 to the first access of m in u_i block in Figure 3(a). Consider each link on the u_i path, say from u_i to u_{i+1}. If the link length is not exactly k, then we are done. If the length is k, and some v_j happens in between, the from the insertion lemma, the k-link can be divided into two shorter links by moving v_j. If no v_j happens between u_i and u_{i+1}, there must exist v_j and v_{j+1} that include u_i and u_{i+1} in between. Since the volume distance from u_i to u_{i+1} is k, v_j and v_{j+1} must appear between u_{i-1} and u_{i+2}, forming the sequence shown in Figure 3(c). If the volume distance from u_{i-1} to v_j is smaller than k, then using the insertion lemma, the link from v_j to u_{i+1} can be divided into two smaller links by moving u_i, and the link from u_{i+1} to u_{i+2} can be divided into two smaller links by moving v_{j+1}. Similarly we can construct smaller links when the volume distance between v_{j+1} and u_{i+2} is less than k. Otherwise, $u_{i-1}, u_i, u_{i+1}, u_{i+2}$, are exactly k-linked. We continue to consider elements of $v_{j-1}, v_{j-2}, \ldots, v_1$ and $v_{j+2}, v_{j+3}, \ldots, vt$ through similar steps. If we can not get a $k-1$ linked path after examining all elements, it means that the path $1, u_1, \cdots, u_s, m$ are exactly k-linked. This is impossible, since the original linked path goes from the first access of 1 to an access of $m+1$. The last link connecting to the access of $m+1$ must have a length greater than k. A contradiction. ∎

We make two observations. First, the proofs do not assume what, when, and whether a section of data is used. It requires only that a section is used together as a block. In N-body simulation, the interactions are calculated within a radius. In this case, an affinity group cannot cross the boundary of a radius. The proofs assume the Morton order for the convenience of illustration, but

it suffices that all data are traversed in some order. The order may change when the same block is accessed at a different time. Hence the theorems can be extended to general divide-and-conquer algorithms. Second, the proofs are for the existence of k. The variable size of data sections changes the value of k but not the existence of k. Therefore, the affinity hierarchy exists when data sections have an arbitrary size.

Given Theorem 1, a natural question is whether the reference affinity groups in divide-and-conquer algorithms can be efficiently discovered. While the answer requires a systematic study that is beyond the scope of this paper, we note that our initial experiments show good results from recursive matrix multiplication. One reason is that in divide-and-conquer algorithms, the elements of the same affinity group are accessed in a similar order, while the reduction in the NP-complete proof requires data be possibly accessed in all possible orders.

4.1 Use in General-Purpose Programs

The use in general-purpose programs has given strong empirical evidence on the validity of the affinity model. Programs often have a large number of homogeneous data objects such as molecules in a simulated space or nodes in a search tree. Each object has a set of attributes. In Fortran 77 programs, attributes of an object are stored separately in arrays. In C programs, the attributes are stored together in a structure. Neither scheme is sensitive to the locality of the program. A better way is to group attributes based on their reference affinity. The results in this section come from our earlier study [21] unless otherwise noted.

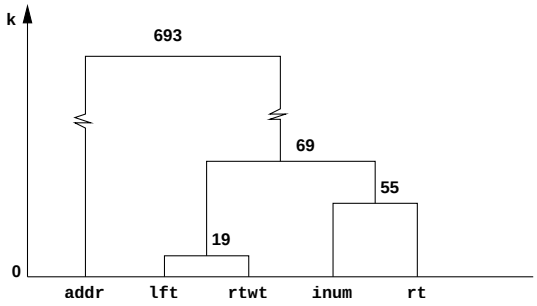


Figure 4: Dendrogram for Cheetah

One example is a splay tree [17] used in a widely distributed cache simulator, *Cheetah*. A tree search consists of a top-down tree search and a bottom-up splay. In a set of executions, the majority of the searches targeted the right-end of the tree, so the program more often follows the right child pointer and uses left rotations. Reference affinity can be efficiently approximated by checking a set of necessary (but not sufficient) conditions. The result is shown by a dendrogram in Figure 4. The fields *lft* and *rtwt* have the closest affinity because of left rotations. The fields *inum* and *rt* have the next closest affinity because of searches to the right. The fifth field, *addr* has the weakest affinity because it is referenced only at the end of a tree search. The number marked at each non-leaf node is the link length of the affinity group. For example, *lft* and *rtwt* are accessed mostly within 19 data elements. The affinity groups have a hierarchical structure.

We improve the object layout through array regrouping in Fortran and structure splitting in C. We tested a range of programs including structural and molecular dynamics simulation, cache simulation and integer sorting. We compared with the layout given by 8 other methods including the programmer, frequency-based models, clustering methods x-means and k-means, and an incomplete affinity model where the linked-path condition was dropped. On average, the affinity-based method improves performance by 5% on an IBM machine and 12% on an Intel PC. What is remarkable is

the consistency. When competing against all 8 other methods for all 9 programs and 2 machines, it ties or wins in 97% of all contests. When measured by the average speed on the two machines, it never loses to the 72 alternative layouts.

Finding the best layout is difficult because the number of choices is exponential to the number of attributes in a object. A program with 14 arrays has a choice of 6 million different layouts. The empirical fact that reference affinity always picks the best layout among all methods shows that it is the most accurate locality model known. The empirical data also shows that the linked-path condition of the affinity definition is critical, even though it is unintuitive. As shown in the complexity proof, it is the main cause of the NP-hardness. However, the study also shows that reference affinity can be efficiently analyzed in large, real programs through approximation.

5 Related Work

Chatterjee et al. demonstrated the performance advantage of the hierarchical data layout for matrix multiplication, Cholesky factorization, and wavelet transform [4]. Mellor-Crummey et al. studied the blocked sparse data layout for irregular programs such as N-body and mesh simulation [13]. We show that reference affinity captures the hierarchical data locality in these and other divide-and-conquer programs. One remaining implementation issue is the placement order of the sub-groups within a group. Previous results suggest that the best order is the one with the least overhead [13].

Thabit showed that data packing for a given block size is NP-hard [19]. Kennedy and Kremer gave a general model that includes run-time data transformation (among other techniques) and showed that the problem is NP-hard [9]. Petrank and Rawitz showed that if $P \neq NP$, no polynomial time method can guarantee a data layout whose number of cache misses is within $O(n^{1-\epsilon})$ of that of the optimal data layout, where n is the data size. They showed that using only pair-wise information, no algorithm can guarantee a data layout whose number of cache misses is within $O(k-3)$ of that of the optimal data layout, where k is the size of cache [15]. The reference affinity is a constructive model attempting to characterize opportunities in commonly used programs. Numerous results in the literature show that such opportunities exist.

Bender et al. used the recursive van Emde Boas layout for dynamic search trees and proved the optimality for random searches [2]. For random searches, the reference affinity gives a flat hierarchy for any constant k. Therefore, it cannot give the recursive layout they gave. The extension for variable-distance affinity groups is a subject of an on-going study. Still, reference affinity is more general because it does not require specific input. It can exploit locality in the type of tree searches that exhibit strong reference affinity, for example, a group of tree nodes are always searched together.

In 1970, Mattson et al. defined the volume distance (named LRU-stack distance) and gave an algorithm that measures a volume distance in $O(M)$ time and $O(M)$ space, where M is the data size of the trace. Over the last thirty years, researchers gradually improved the asymptotic complexity to $O(logM)$ time and $O(M)$ space. In 2003, we gave an approximation algorithm that guarantees any given accuracy (e.g. 99.9%) but takes only $O(loglogM)$ time and $O(logM)$ space [7].

6 Summary

In this paper we have presented till now the most general model of hierarchical data locality. The reference affinity is defined based on the volume distance and parameterized by the link length k. We have characterized the complexity of the linking, checking, and finding problems. When k is 1 and 2, we show that all three problems are in P and give a polynomial-time algorithm for finding affinity groups. When k is 3, we show that the linking and checking problems are NP-complete, and the finding problem is NP-hard. Reference affinity has many uses. We prove that it finds the hierarchical locality from the trace of hierarchical computations such as recursive matrix

multiplication, factorization, wavelet transform, N-body and mesh simulation. We cite empirical results showing the reference affinity can be efficiently approximated. The theorems and proofs have established new links between the path-based affinity model, the structure of computation, and the hierarchical locality of data. These results strengthen our understanding of the theoretical relation between computation and data and open the door for research in more effective and efficient programming methods for general-purpose programs on modern computer systems.

Acknowledgments

This material is based upon work supported by a grant from The National Science Foundation grant numbers E1A-0080124 and EIA-0205061 (subcontract , Keck Graduate Institute), and Department of Energy award #DE-FG02-02ER25525. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of above named organizations.

References

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the ACM Conference on Theory of Computing*, New York, NY, 1987.

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

[3] K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, Paderborn, Germany, August 2002.

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

[5] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[6] P. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), January 1980.

[7] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[8] G. Hunt and M. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99)*, Orleans, Lousiana, February 1999.

[9] K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4), 1998.

[10] D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.

[11] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*, New York City, NY, June 2004.

[12] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

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

[14] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[15] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.

[16] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2), 1985.

[17] D. D. Sleator and R. E. Tarjan. Self adjusting binary search trees. *Journal of the ACM*, 32(3), 1985.

[18] A. J. Smith. Cache Memories. *Computing Surveys*, 14(3), September 1982.

[19] K. O. Thabit. *Cache Management by the Compiler*. PhD thesis, Dept. of Computer Science, Rice University, 1981.

[20] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2), 2001.

[21] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

A Proofs

Theorem 1 **For each $k \geq 3$, Pw-k-Aff is NP-complete.**

Proof It is obvious that the problem is in NP. We will prove its NP-hardness by constructing a polynomial-time many-one reduction to Pw-k-Aff from 3-SAT, which is the problem of testing, given a formula of conjunctive normal form in which each clause has at most 3 literals, whether the formula is satisfiable. We consider the variant of this problem in which each variable appears as a literal (positively or negatively) at most three times. This problem is also known to be NP-complete (see, e.g., [14]). Without loss of generality, we can assume that all variables appear both positively and negatively in the formula. If a variable appears only positively (respectively, negatively) then we can create a simpler, equivalent formula by setting the value of the variable to *true* (respectively, *false*).

Let φ be a CNF formula of N variables and M clauses in which each clause has at most 3 literals and each variable appears at most three times. Let $x_1, \ldots, x_N$ be the variables of φ and $C_1, \ldots, C_M$ be the clauses of φ.

Let λ_{in} and λ_{out} be two distinct labels. We will define a sequence T whose first label is λ_{in} and whose last label is λ_{out}. λ_{out} appears nowhere else in the sequence. We will consider the problem of creating a k-linked path between the two. The sequence is of the form

$$\lambda_{in}\Sigma\Gamma_1 \cdots \Gamma_N\Theta_1 \cdots \Theta_M\lambda_{out},$$

.

The sequence Σ is the k repetitions of $\nu_1 \cdots \nu_k$ separated by $k-1$ λ_{in}'s. where $\nu_1, \ldots, \nu_k$ are k pairwise distinct labels. Recall that for a pair of positions to be k-linked there must be a set of intermediate points with pairwise distinct labels in which the reuse distance between each neighboring intermediate points is at most k. To create such a path between our two end points, the subsequence Σ must be traversed without visiting a same label more than once so that the distance between the two neighboring visited points have reuse distance at most k. The only way to construct such a path is to visit every $(k+1)^{st}$ element of Σ besides the first λ_{in}, exiting at the first element after Σ. This path visits $\nu_1, \ldots, \nu_k$ exactly once. This means that any k-link path between our two endpoints should not visit any one of $\nu_1, \ldots, \nu_{k+1}$ again.

For each x_i appeared in the formula, $1 \leq i \leq N$, Γ_i if of form

$$\alpha_{i,1}\gamma_{i,1}\gamma_{i,2}\gamma_{i,3}\nu_1 \cdots \nu_{k-2}\gamma_{i,1}\alpha_{i,2}\nu_1 \cdots \nu_{k-1}.$$

The α's here appear nowhere else in the sequence. Each γ appears at most once elsewhere. If it does indeed, it appears in one of the Θ's. Suppose that a k-linked path between the two endpoints lands on $\alpha_{i,1}$. Then the path can only be threaded in Γ_i using one of the following paths:

1. $[\gamma_{i,3}, \alpha_{i,2}]$,

2. $[\gamma_{i,3}, \gamma_{i,1}, \alpha_{i,2}]$,
3. $[\gamma_{i,2}, \gamma_{i,3}, \alpha_{i,2}]$,
4. $[\gamma_{i,2}, \gamma_{i,3}, \gamma_{i,1}, \alpha_{i,2}]$,
5. $[\gamma_{i,2}, \gamma_{i,1}, \alpha_{i,2}]$,
6. $[\gamma_{i,1}, \gamma_{i,2}, \gamma_{i,3}, \alpha_{i,2}]$, and
7. $[\gamma_{i,1}, \gamma_{i,3}, \alpha_{i,2}]$.

Consider the set of all γ's that has not been visited yet. The set is

1. $\{\gamma_{i,1}, \gamma_{i,2}\}$,
2. $\{\gamma_{i,2}\}$,
3. $\{\gamma_{i,1}\}$,
4. $\emptyset$,
5. $\{\gamma_{i,3}\}$,
6. $\emptyset$,
7. $\{\gamma_{i,2}\}$,

Two crucial observations here are that (a) there is no set that contains $\gamma_{i,3}$ and one extra element and (b) that the first set has both $\gamma_{i,1}$ and $\gamma_{i,2}$. Suppose that x_i appears three times in the formula, twice as x_i and once as $\overline{x_i}$. Then we use $\gamma_{i,1}$ to denote the first occurrence of x_i, $\gamma_{i,2}$ to denote the second occurrence of x_i, and $\gamma_{i,3}$ to denote $\overline{x_i}$. In the case when $\overline{x_i}$ appears twice and x_i appears once, we use $\gamma_{i,1}$ to denote the first occurrence of $\overline{x_i}$, $\gamma_{i,2}$ to denote the second occurrence of $\overline{x_i}$, and $\gamma_{i,3}$ to denote x_i. In the case when both x_i and $\overline{x_i}$ appear only once, we use $\gamma_{i,1}$ to denote the unique occurrence of x_i and $\gamma_{i,3}$ to denote the unique occurrence of $\overline{x_i}$. Note that all of these possible paths must land the first element after Γ_i.

For each i, $1 \leq i \leq M$, such that C_i has exactly two literals, Θ_i is of the form

$$\beta_{i,1}\theta_{i,1}\theta_{i,2}\nu_3 \cdots \nu_k.\beta_{i,2}\nu_2 \cdots \nu_k,$$

and for each i, $1 \leq i \leq M$, such that C_i has exactly three literals, we construct Θ_i as

$$\beta_{i,1}\theta_{i,1}\theta_{i,2}\theta_{i,3}\nu_4 \cdots \nu_k.\beta_{i,2}\nu_2 \cdots \nu_k,$$

where $\theta_{i,l}$ is the l^{th} literal of C_i. Note here that the literals in the clause are replaced using γ's in the sequence according to the rules in the construction of Γ's. Suppose that the k-linked path between our two endpoints land on $\beta_{i,1}$. Since there are k labels between $\beta_{i,1}$ and $\beta_{i,2}$ and none of the ν's can be visited again, the k-linked path can only be extended if one of the θ literals is visited. The segment after $\beta_{i,2}$ forces the path to land on the element right after Θ_i.

we can see that Σ is of length $k(k+1)-1$, for each i, $1 \leq i \leq N$, Γ_i has length $2k+3$, and for each i, $1 \leq i \leq M$, Θ_i has length $2k+1$. So, the total length of the sequence, including the two endpoints, is

$$2 + k(k+1) - 1 + N(2k+3) + M(2k+1),$$

which is equal to $k(2N+2M+k+1)+3N+M+1$, which is polynomial of the size of the CNF formula. So the construction can be done in polynomial time.

We view the literals that are visited in Θ_i as those satisfied by the assignment represented by the path. For such a path to be valid, the selections in the Θ sections have to be made so that the literals satisfying the clauses are still available. Suppose that φ is satisfiable. Let A be a satisfying assignment of φ. Construct the path within Θ's so that the those that are visited are precisely those that are satisfied by A. Then it is possible to select the paths in Γ so that none of those visited in Θ are visited in Γ. So, the two endpoints are k-lined.

On the other hand, suppose that φ is not satisfiable. Take any potentially k-linked path π in the Θ's. There exist at least one variable, x_i for which both one occurrence of x_i and one occurrence of $\overline{x_i}$ is selected.

Then it is not possible to construct a k-linked path within Γ_i, so there is no k-linked path between the two endpoints.

We note here that the set of labels, Λ, which is the part of the instance is the set of all labels that we've defined. By now, we have constructed a polynomial-time many-one reduction from 3-SAT to Pw-k-Aff. Since Pw-k-Aff apparently belongs to NP, we prove that Pw-k-Aff is a NP-complete problem. ∎

Corollary 1 **For $k \geq 3$, the problem of checking reference affinity groups is NP-complete.**

Proof Suppose the group of data elements is G. First, let's show that this problem belongs to NP. This can be done by first guessing the possible supersets of G, say G'. For every two different data elements $x, y \in G'$, for every a_x, we guess it can be connected to the nearest a_y located left-side or right-side, and then we guess a link-path between them and then verify if this is a link path of link-length k. If it is, then continue to check other a_x's and then other pairs of data elements. But if not, it will just refuse to accept. We can check for all of the pairs and all accesses of x in a sequential way. If every pairs and every accesses are checked to be linked successfully, then accept.

By the definition of reference affinity group, for any $x, y \in G$, for all a_x, we need to check if there exists an a_y, such that a_x and a_y are k-linked. The only way is to check if there is a k-linked path from a_x to the left-side or right-side nearest a_y. So we can see that if there is a polynomial-time algorithm for checking reference affinity problem, then there is a polynomial-time algorithm for Pw-k-Aff problem. Thus we have proved that for $k \geq 3$,checking reference affinity group problem is NP-complete problem. ∎

Corollary 2 **For $k \geq 3$, the problem of finding reference affinity groups is NP-hard.**

Proof The proof is quite straightforward. If there is a polynomial-time solution that can find out the reference affinity groups, then we can solve the problem of checking reference affinity groups in polynomial time. This contradicts with Corollary 1. ∎

Theorem 2 **For $k = 2$, Pw-k-Aff is NL-complete.**

Proof *2*-CNF-SAT is the problem of testing whether a given conjunctive normal form formula with two literals per clause is satisfiable. This problem is the standard NL-complete problem. By following the proof of Theorem 1 with $k = 2$, we can show that the *2*-CNF-SAT is reducible to Pw-k-Aff for $k = 2$.

To prove that $PWkAff$ belongs to NL for $k = 2$, suppose that a set of labels Λ, a sequence $\Sigma = \{\sigma_i\}_{i\geq 1}^{M}$ over Λ, an integer $k \geq 0$, and two integers I and J, $1 \leq I \leq J \leq M$ are given as an instance to the problem. We wish to test whether I and J are k-linked.

Since the elements before the I^{th} entry and those after the J^{th} are irrelevant to the problem at hand, we may assume, Without loss of generality, that $I = 1$ and $J = M$. Also, if the i^{th} entry and the $(i+1)^{st}$ entry are the same, at most one of the two can be visited, and if one is visited at all which one doesn't matter. So, one of them can be safely removed. This means that, for all i, $2 \leq i \leq M-2$, $\sigma_i \neq \sigma_{i+1}$.

For each i, $2 \leq i \leq M-1$, let y_i be the variable that represents whether the i^{th} element is visited. We construct a formula φ by joining the following size-two clauses:

- for each i, $2 \leq i \leq M-2$, $(y_i \vee y_{i+1})$, and
- for all $\rho \in \Sigma$ and for all i and j such that $2 \leq i < j \leq M-1$ and $\sigma_i = \sigma_j = \rho$, $(\overline{y_i} \vee \overline{y_j})$.

Suppose that this formula is satisfiable. Let A be a satisfying assignment of the formula. Then A clearly defines a k-linked path, since only those belonging to Σ are visited, no element in Σ is visited more than once, and there is at most one entry between any two neighbors on the path. Similarly, if there is a k-linked path, then by setting the truth value of each variable according to whether the node is included in the path, we can satisfy the formula. So, the satisfiability of the formula is equivalent to the existence of a k-linked path. ∎

Theorem 3 **For $k = 2$, the problem of finding reference affinity groups is in P.**

Algorithm 1 can be found in Section 3. Here we present the detailed proof.

Proof First let us show this is a polynomial-time algorithm. By Theorem 2, we need polynomial time to test whether two data accesses are 2-linked. Hence, testing if two data elements is 2-linked with respect to a given group can be done in polynomial time. Constructing the graph G needs only polynomial time. For the reference affinity group that x belongs to, we remove at most m data elements from the group, where m

is the number of data elements in the trace. There are at most m reference affinity groups. Therefore, the algorithm takes polynomial time.

Next we prove the correctness. First, it is easy to see that the groups found by this algorithm satisfy the first condition of reference affinity. Second, let us show every group is the maximal size possible. We show that the algorithm removes z correctly. Removing z at step 7 is straightforward. The correctness of the removal of z at step 10 can be proved by contradiction. Suppose z and x belong to the same group G_1. We have $y \notin G_1$. From the algorithm, an access a_y cannot be 2-linked to any access of z. Since x and y are 2-linked, there are some accesses of x that is 2-linked to a_y. We pick the nearest one as a_x. Without loss of generality, we assume a_x appears at the right side of a_y. Similarly, we choose a_z, which is 2-linked to and nearest to the a_x. This a_z can not appear on the left side of a_x. Otherwise, we have two cases. First, if a_z appears between a_y and a_x, then the path from a_y to a_x must pass the very data element at the right side of a_z, since $k = 2$. Then the a_y can be 2-linked to this a_z by replacing the very data element with a_z, which is a contradiction. Second, if a_z appears on the left side of a_y, since x and z are in the same group, a path exists from a_x to a_z without passing a_y. This path must land on the very data element at the right side of a_y, since $k = 2$. Then we can replace the very data element with a_y and get a new path from a_y to a_z, which is also a contradiction.

Now let's select the leftmost data element in G_1 that appears on the section of trace between the a_y and a_z. Suppose it is a_l. This is shown in Sequence (2).

$$...y...l...x...z... \tag{2}$$

We first show that a path exists from a_y to a_l with respect to $(G - G_1) \bigcup \{l\}$. Since a_y is 2-linked to a_x with respect to group G, there is a path π connecting them. If π does not pass a_l, it must pass the very data element at the left side of a_l, since $k = 2$. A new path $\pi_1$can be generated from a_y to a_l by first reaching the very data element and then one step further to a_l. If π passes a_l, then we pick the segment from a_y to a_l as π_1. All of the data elements on the path π_1 is in $G - G_1$ except for l.

Since l is in the same group with z, there is a path π_2 from a_l to a_z with respect to G_1. We get a new path π' by merging paths π_1 and π_2. Now π' is a 2-linked path without duplicated data elements from a_y to a_z, which is a contradiction with step 9. ∎

Theorem 5 **For $k = 1$, there is a polynomial-time solution for finding reference affinity groups.**

Algorithm 2 Finding reference affinity groups when k=1

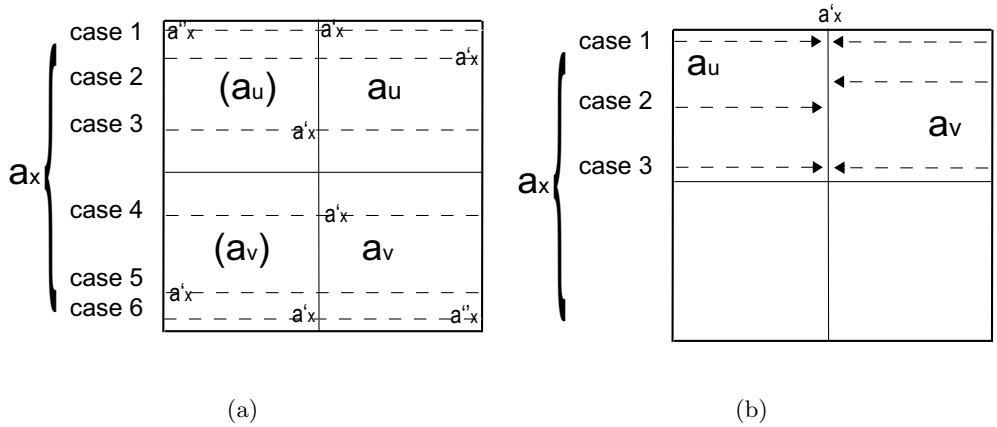
procedure *FindReferenceAffinityGroup_1(T)*

1: {*T is the trace,* $k = 1$}
2: *encode the data elements according to the order of appearance in the trace. Suppose there are m distinct data elements.*
3: ***while*** *there exist ungrouped codes* ***do***
4: *pick the smallest ungrouped datum as s.*
5: ***for*** *t=m to s step* -1 ***do***
6: ***if*** *IsAGroup(T,s,t)* ***then***
7: **break**;
8: ***end if***
9: ***end for***
10: *output elements in* $\{s, \ldots, t\}$ *as a group.*
11: ***end while***

endFindReferenceAffinityGroup_1

procedure *IsAGroup(T,s,t)*

1: ***for*** *i varies from* 1 *to* $|T|$ ***do***
2: ***if*** *T[i] is within s and t* ***then***
3: ***if*** *The elements T[i] can be* 1*-linked to with respect to* $\{s, \ldots, t\}$ *can not cover set* $\{s, \ldots, t\}$ ***then***
4: **return** *false;*
5: ***end if***



(a) (b)

Figure 5: Cases in proving the insertion lemma

6: ***end if***
7: ***end for***
8: **return** *true;*
endIsAGroup

It is straightforward to show that the algorithm is polynomial time and can output the correct reference affinity groups.

Lemma 1 **Given two different data elements u and v; their accesses a_u and a_v where the volume distance from a_u to a_v is exactly k; and a third access a_x, which happens between a_u and a_v in the trace; then there exists an access a'_x between a_u and a_v such that the volume distance from a_u to a'_x is less than k, and the volume distance from a'_x to a_v is less than k.**

Proof The element u is either earlier or later than v in the data space. Because a link and a path are not directed, the two cases are symmetrical. Without loss of generality, we assume u is before v. Consider the smallest m-block that contains u, v, a_u, a_v. The element x must be in the data section of the block; otherwise the path from a_u to a_v does not go through x. There are two cases shown by the two graphs in Figure 5, each has six sub-cases. The location of a_x is given for each sub-case in the figure. In most cases, a_x splits the k-link from a_u to a_v into two shorter links of less than k. The first case is when a_u and a_v are in upper and lower half blocks. In the first sub-case of the first case, we need to use one of the two locations, marked by a'_x and a''_x. Then we use the a_u to break the link from the access of x to a_v and treat the access to x as a_u. The last sub-case of the first case is similar. The second case happens when a_u and a_v are both in the upper or lower half block. Figure 5 shows the three out of the six sub-cases when both accesses are in the upper half block. The other three sub-cases are symmetrical. In sub-case 1 and 3, we pick a'_x to be in the middle on the same side of a_x. In sub-case 2, we use one of the two middle points depending on the position of a_x. ∎