Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity

Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding

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

ABSTRACT

While the memory of most machines is organized as a hierarchy, program data are laid out in a uniform address space. This paper defines a model of *reference affinity*, which measures how close a group of data are accessed together in a reference trace. It proves that the model gives a hierarchical partition of program data. At the top is the set of all data with the weakest affinity. At the bottom is each data element with the strongest affinity. Based on the theoretical model, the paper presents *k-distance analysis*, a practical test for the hierarchical affinity of source-level data. When used for array regrouping and structure splitting, k-distance analysis consistently outperforms data organizations given by the programmer, compiler analysis, frequency profiling, statistical clustering, and all other methods we have tried.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization, compilers*

General Terms

Algorithms, Languages, Performance

Keywords

Program locality, program transformation, reference affinity, volume distance, reuse signature, array regrouping, structure splitting

1. INTRODUCTION

All current PCs and workstations 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.

PLDI'04, June 9–11, 2004, Washington, DC, USA.

To improve cache utilization we need to group related data into the same cache block. The question is how to define the relation. We believe that it should meet three requirements. First, it should be solely based on how data are accessed. For example in an access sequence "$abab..ab$", a and b are related and should be put in the same cache block, regardless how they are allocated and whether they are linked by pointers. Second, the relation must give a unique partition of data. Consider for example the access sequence "$abab..ab...bcbc..bc$". Since data a and c are not related, b cannot relate to both of them because it cannot stay in two locations in memory. Finally, the relation should be a scale. Different memory levels have blocks of increasing sizes, from a cache block to a memory page. The grouping of "most related" data into the smallest block should precede the grouping of "next related" data into larger blocks. In summary, the relation should give a unique and hierarchical organization of all program data.

In this paper, we define such a relation we call *reference affinity*, which measures how close a group of data are accessed *together* in an execution. Unlike most other program analysis, we measure the "togetherness" using the *LRU stack distance*, defined as the amount of data accessed between two memory references in an execution trace [28]. As a notion of locality, stack distance is bounded, even for long-running programs. The long stack distance often reveals long-range data access patterns that may otherwise hide behind complex control flows, indirect data access, or variations in coding and data allocation. We prove that the new definition gives a unique partition of program data for each distance k. When we decrease the value of k, the reference affinity gives a hierarchical decomposition and finds data sub-groups with closer affinity, much in the same way we sharpen the focus by reducing the radius of a circle.

Two earlier studies defined a *reuse signature* as a histogram of the reuse distance of all program data access [17, 36]. They showed that the reuse signature has a consistent pattern across all data inputs even for complex programs or regular programs after complex compiler optimizations. This suggests that we can analyze the reference affinity of the whole program by looking at its reuse signatures from training runs.

We present *k-distance analysis*, which simplifies the requirements of reference affinity into a set of necessary conditions about reuse signatures. The simplified conditions can then be checked efficiently for large, complex programs. The parameter k has an intuitive meaning—elements in the same group are almost always used within a distance of k data elements. The analysis handles sequential programs with arbitrarily complex control flows, indirect data access, and dynamic memory allocation. The analysis uses

multiple training runs to take into account the variation caused by program inputs. In addition, we will use the analysis to find the worst reference affinity and measure the range of the impact from different data layouts.

Although reference affinity and k-distance analysis have strong properties, they are not optimal. We will formulate the problems in terms of partial and dynamic reference affinity and discuss their complexity. Our analysis checks the necessary rather than the sufficient conditions of reference affinity, so it may include data with false affinity. We will show that the probability of error is small and can be further reduced by strengthening the conditions.

The rest of the paper is organized as follows. Section 2 defines the formal model of reference affinity and proves its properties. Section 3 presents k-distance analysis and a comparison with a number of other methods. Section 4 describes the compiler support for array regrouping and structure splitting. The last three sections present the experimental evaluation, related work, and conclusions.

2. MODEL OF REFERENCE AFFINITY

This section first defines three preliminary concepts and gives two examples of the reference affinity model. Then it presents the formal definition and proves the properties including the unique and hierarchical partition of program data.

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.

The *LRU stack 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 let $dis(a_x, a_y) = dis(a_y, a_x)$. If $T[a_x] = T[a_y]$, $dis(a_x, a_y) = 0$. The distance is the volume of data accessed between two points of a trace, so we also call it the *volume distance*. In comparison, the time distance is the difference between the logical time of two accesses. For example, the volume distance between the accesses to a and c in the trace $abbbc$ is 2, while the time distance is 4. 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.

Based on the volume distance, we define a *linked path* in a trace. It 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}, ..., 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 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 linked path from a_x to a_y with link length k for the set S.

We now explain reference affinity with two example address traces in Figure 1. The "..." represents accesses to data other than w, x, y, and z. In the first example, accesses to x, y, and z are in three sections. The three elements belong to the same affinity group because they are always accessed together. The consistency is important for data placement. For example, x and w are not always used together, so putting them into the same cache block would waste cache space when only one of the two is accessed. The example shows that finding this consistency is not trivial. The accesses to the three data elements appear in different orders, with a different frequency, and mixed with accesses to other data. However, one property holds in all three sections — the accesses to the three elements are connected by a linked path with the link length 2.

```
xyz ... xwzzy ... yzvvvvvx ...
```

(1) The affinity group {x,y,z} with link length k = 2

```
wxwxuyzyz ... zyzyvwxwx ...
```

(2) The affinity group {w,x,y,z} at k=2 becomes two groups {w,x} and {y,z} at k=1

Figure 1: Examples of reference affinity model and properties

As we will prove later, affinity groups are parameterized by the link length k, and they form a partition of the program data for each k. The second example in Figure 1 shows that this group partition has a hierarchical structure. The affinity group with the link length of 2 is $\{w, x, y, z\}$. If we reduce the link length to 1, the two new groups will be $\{w, x\}$ and $\{y, z\}$. The groups at a smaller link length are subsets of the groups at a greater link length. The hierarchical structure is useful in data placement because it can match different-size affinity groups to a multi-level cache hierarchy.

We now present the formal definition of the reference affinity.

DEFINITION 1. **Strict Reference Affinity**. *Given an address trace, a set G of data elements is a strict 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*

The following theorem proves that strict affinity groups are consistent because they form a partition of program data. In other words, each data element belongs to one and only one affinity group.

THEOREM 1. *Given an address trace and a link length k, the affinity groups defined by Definition 1 form a unique partition of program data.*

PROOF. We show that any element x of the program data belongs to one and only one affinity group at a link length k. For the "one" part, observe that Condition (1) in Definition 1 holds trivially when x is the only member of a group. Therefore any element must belong to some affinity group.

We prove the "only-one" part by contradiction. Suppose x belongs to G_1 and G_2 ($G_1 \neq G_2$). Then we can show that $G_3 = G_1 \cup G_2$ satisfies Condition (1).

For any two elements $y, z \in G_3$, if both belong to G_1 and G_2, then Condition (1) holds. Without loss of generality, assume $y \in$

$G_1 \wedge y \notin G_2$ and $z \in G_2 \wedge z \notin G_1$. Because $y, x \in G_1$, any a_y, must have a linked path to an a_x, that is, there exist $y_1, \ldots, y_t \in G_1$ and an access a_x such that $dis(a_y, a_{y_1}) \leq k \wedge \ldots \wedge dis(a_{y_t}, a_x) \leq k$. Similarly, there is a linked path for this a_x to an a_z because $x, z \in G_2$, that is, there exist $z_1, \ldots, z_m \in G_2$ and an access a_z such that $dis(a_x, a_{z_1}) \leq k \wedge \ldots \wedge dis(a_{z_m}, a_z) \leq k$.

If $y_1, \ldots, y_t \notin \{z_1, \ldots, z_m\}$, then there is a linked path from a_y to some a_z. Suppose $y_1, \ldots, y_{i-1} \notin \{z_1, \ldots, z_m\}$ but $y_i = z_p$. Then we have a linked path from a_y to a_{y_i}. Since $y_i = z_p \in G_2$, there is a linked path from y_i to z, that is, there exist $z'_1, z'_2, \ldots, z'_n \in G_2$ such that $dis(a_y, a_{y_1}) \leq k \wedge \ldots \wedge dis(a_{y_{i_1}}, a_{y_i}) \leq k \wedge dis(a_{y_i}, a_{z'_1}) \wedge \ldots \wedge dis(a'_{z_n}, a_z) \leq k$. Now y_i belongs to $G_1 \cap G_2$, just like x. We have come back to the same situation except that the linked path from a_y to a_{y_i} is shorter than the path from a_y to a_x. We repeat this process. If $y_1, \ldots, y_{i-1} \notin \{z'_1, \ldots, z'_n\}$, then we have a linked path from a_y to a_z. Otherwise, there must be $y_j \in \{z'_1, \ldots, z'_n\}$ for some $j < i$. The process cannot repeat forever because each step shortens the path from y to the access chosen next by this process. It must terminate in a finite number of steps. We then have a linked path from a_y to a_z in G_3. Therefore, Condition (1) always holds for G_3. Since $G_1, G_2 \subset G_3$, they are not the largest sets that satisfy Condition (1). Therefore, Condition (2) does not hold for G_1 or G_2. A contradiction. Therefore, x belongs to only one affinity group, and affinity groups form a partition.

For a fixed link length, the partition is unique. Suppose more than one type of partition can result from Definition 1, then some x belongs to G_1 in one partition and G_2 in another partition. As we have just shown, this is not possible because $G_3 = G_1 \cup G_2$ satisfies Condition (1) and therefore neither G_1 nor G_2 is an affinity group. □

As we just proved, reference affinity is consistent because all members will always be accessed together (i.e. linked by some linked path with the link length k). The consistency means that packing data in an affinity group will always improve cache utilization. In addition, the group partition is unique because each data element belongs to one and only one group for a fixed k.

Next we prove that the strict reference affinity has a hierarchical structure — an affinity group with a shorter link length is a subset of an affinity group with a greater link length.

THEOREM 2. *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'.*

PROOF. We show that any affinity group at k is a subset of some affinity group at k'. Let G be an affinity group at k and G' be the affinity group at k' that overlaps with G ($G \cap G' \neq \emptyset$). Since any $x, y \in G$ are connected by a linked path with the link length k, they are connected by a linked path with the larger link length k'. According to the proof of Theorem 1, $G \cup G'$ is an affinity group at k'. G must be a subset of G'; otherwise G' is not an affinity group because it can be expanded while still guaranteeing Condition (1). □

Finally, we show that elements of the same affinity group are always accessed together. When one element is accessed, all other elements will be accessed within a bounded volume distance.

THEOREM 3. *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.*

PROOF. According to Definition 1, for any y in G, there is a linked path from a_x to some a_y. Sort these accesses in time order. Let a_w be the earliest and a_v be the latest in the trace. There is a linked path from a_w to a_x. Let the sequence be $a_{x_1}, a_{x_2}, \ldots, a_{x_t}$. The volume distance from a_w to a_x is $dis(a_w, a_x)$. It is no greater than $dis(a_w, a_{x_1}) + dis(a_{x_1}, a_{x_2}) + \ldots + dis(a_{x_t}, a_x)$, which is $(t+1)k \leq |G|k$. The bound of the volume distance from a_x to a_v is the same. Considering that a_v is included in the section, the total volume distance is at most $2k|G| + 1$. □

The strict affinity requires that members of an affinity group be always accessed together. Ding and Kennedy showed that it gives the best data layout when no side effect (i.e. increased cache misses) is allowed [14]. On most machines, it is still profitable to group data that are almost always accessed together because the side effect would not outweigh the benefit. For programs with different behavior phases, it may be profitable to exploit reference affinity in each phase and change data layout between phases. We call these extensions the *partial reference affinity* and the *dynamic reference affinity*. Ding and Kennedy showed that the optimal data layout in these two cases is machine dependent and finding the optimal layout is an NP-hard problem [14]. Next we present a method that measures the "almost strict" reference affinity in complex programs. The link length k will play a critical role as it did in this section.

3. WHOLE-PROGRAM AFFINITY

We now study the reference affinity among the source-level data. We specifically target data arrays and instances of structure fields because they account for major portions of global and heap data in most programs. Since an array or a field represents a set of data, we need to extend the affinity definition. The affinity exists among data sets if the sets have the same number of elements, and one element in one set has the reference affinity with one and only one element in every other set. In particular, reference affinity exists for two arrays, A and B, if the reference affinity exists for $A[i]$ and $B[i]$ for all i. The affinity exists for structure fields, f_1 and f_2, if it exists for $o.f_1$ and $o.f_2$ for all instance o. Next, we introduce the reuse signature as the basis for whole-program affinity analysis. We then present k-distance analysis and a comparison with other methods.

3.1 Reuse signature

The reuse signature of a set of data is the histogram of the reuse distance of their accesses. Ding and Zhong showed that the whole-program reuse signature exhibits a consistent pattern across data inputs for large, complex integer and floating-point benchmarks [17]. The result suggests that we can use the reuse signature from one profiling run to infer the reuse signatures in other executions. Therefore, reuse signature allows not just the whole-trace analysis but also the whole-program analysis of reference affinity.

We illustrate the use of reuse signature through an example program, *Cheetah*, a fully associative LRU cache simulator that is part of the SimpleScalar 3.0 tool set. The main data structure is a splay tree, and each tree node has a number of fields, of which we concern three in this example. Based on a profile from a simple input, we draw the accesses to the three fields on time-space graphs shown in Figure 2. Each access is a point whose x-axis value is the logical time (in memory references) and y-axis the memory address. The similarity of the graphs suggests that the two fields *rtwt* (the sub-tree weight) and *lft* (left-child pointer) have the reference affinity because they seem to be always accessed together. The third

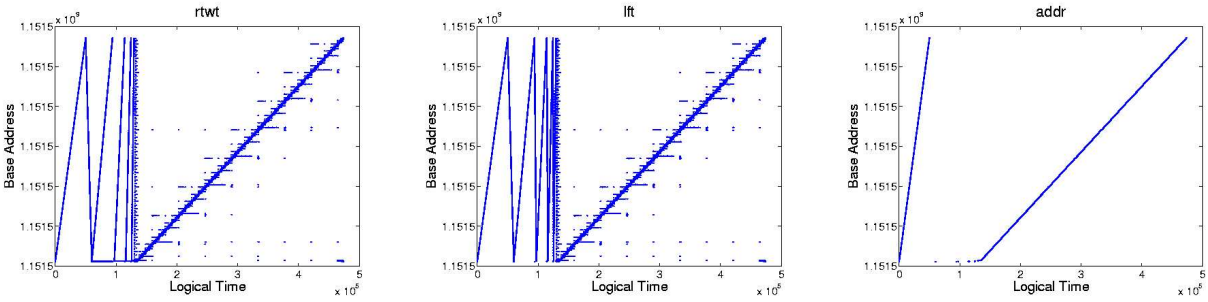

Figure 2: The time-space graphs of the accesses to the three tree-node fields

field, *addr*, is accessed only occasionally together with the other two fields. A manual inspection of the splay-tree algorithm confirms these conjectures. The program uses the first two fields together for the tree rotation at every step of a tree search, while it uses the third field only at the end of a tree search (the tree is indexed by time not the address).

Figure 3 shows the reuse signature of the three fields. Each is a histogram of the reuse distance for all accesses to a structure field. The reuse distance of an access is the volume distance between this and the previous access to the same data element. The x-axis is a sequence of bins representing different ranges of a reuse distance. The bins may be of the same size (linear scale), exponential size (log scale as in this example), or their combinations. The y-axis gives the number of memory accesses whose reuse distance falls into each bin. The figure shows reuse signatures for reuse distances greater than 1024. The short-distance reuses (that are not shown) account for about 70% of accesses to *addr* and over 85% to *lft* and *rtwt*.

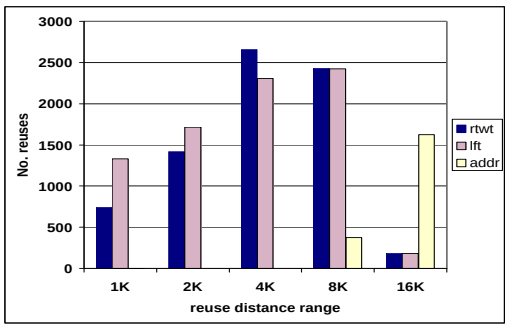

Figure 3: The reuse signatures of the three tree-node fields

We now compare the reuse signature of *rtwt* and *lft*. The latter has more reuses at the first bin because it is repeatedly accessed during initialization. However, the additional reuses have a short distance and do not affect reference affinity. The number of reuses in the second and third bins also differ. This is because many reuses with a distance close to 4K are separated by the arbitrary bin boundary. The total number of reuses with a distance more than 2K is very similar. In the last two (larger) bins, *rtwt* and *lft* have the same number of accesses, but the number is different for *addr*.

We use the reuse signature for affinity analysis as follows. We treat reuse signatures as vectors, remove the first few elements (reuse distances shorter than 2048), and find affinity groups by comparing the reuse signatures. The comparison method is key to the affinity analysis, as described next. Reuse signatures are measured through profiling, so a remaining question is whether other inputs to *Cheetah* have the same affinity. As a cache simulator, the access pattern in *Cheetah* depends on the input. After checking a few other inputs, we found that the consistency remains: the time-space graph and the upper portion of the reuse signature remain similar between *rtwt* and *lft* but different from *addr*. In fact, we can consider multiple training runs by combining their reuse signatures and therefore rule out false affinity relations that appear in one input but not others. As shown by this example, we may identify similar access patterns even if they are hidden behind branches, recursive functions, and pointer indirections.

We have converted the problem from checking reference affinity in a trace, to checking patterns in time-space graphs, and finally to checking the similarity of reuse signatures. The compression of the information is dramatic: from billions of memory accesses and trillions of time-space points to vectors of $logM$ elements, where M is the size of program data (the maximal volume distance). Next we inject rigor into the use of the reuse signature by formulating it in terms of the necessary conditions of the reference affinity.

3.2 K-distance analysis

To measure the reference affinity, we check a necessary but not sufficient condition on the reuse signature. We first derive the necessary condition for two data elements, then extend it to two or more data sets, and discuss the improvements to the condition. K-distance analysis checks an improved version of the necessary condition. The derivation of the necessary condition is complex. The following description is terse for lack of space. A reader may skip to Equation 1 for the formula used by the k-distance analysis.

As a basic case, let an affinity group have two elements, x and y, always accessed within a link length k. For any access a to one of the two elements, there must be an access a' to the other element, within the volume distance k. We call the smallest section of the trace that contains a and a' an affinity window. We merge two windows by taking their union whenever they overlap in time, so the resulting windows are disjoint. By now all accesses to x and y are within one of the windows. We further divide these accesses into two groups. An access is a local reuse if the previous access is inside the same window. Otherwise, it is a remote reuse. We visualize a reuse distance by an edge connecting the two accesses in the trace. The edges are either local (inside a window) or remote (between two closest windows).

We can now pick the threshold for removing the short reuse distances. For reasons we do not elaborate, the necessary condition would be too weak to be useful if we are not careful in setting the threshold. The threshold must be such that after the removal, the

ending points of the remaining long distances (accesses to x and y) still have the reference affinity among themselves. The threshold, h, must meet two conditions. It must be no less than $2k$, so that we remove all local edges. In addition, no reuse distance should be close to h. To be exact, for an integer range w that is at least $2k$ in size and contains h inside the range, we require that no reuse distance be within the range. The second condition ensures that the remote edges between two windows are either both below h or both over h. Then the ending points of the remaining reuse distances have the reference affinity among themselves. Since the threshold should be lower than the length of long reuse distances, the range w may not always exist. However, as Ding and Zhong observed in most programs they tested, a reuse distance either stays constant or lengthens as the program input size increases [17]. In these programs, we can always find the range by using a large enough input in the analysis.

After removing short reuse distances, x and y have the same number of remaining long reuse distances. They are paired. If there is a long reuse distance of one element between two windows, there is a long reuse distance of the other element between the same two windows. The maximal length difference between each pair is $2k$. We do not include the proof, but the key observation in the proof is as follows. The ending point of a long reuse distance is the first access of the element in the window, while the starting point of the long distance is the last access of the element in the preceding window. Since the maximal difference between each pair is $2k$, a *necessary condition* for the reference affinity relation between x and y is that after removing short reuse distances, the average length of their long reuse distances is no more than $2k$.

For two sets of data, X and Y, to have the reference affinity, the necessary condition holds between the affinity pairs. Therefore, a necessary condition is that the average length of the long reuse distances from the two sets differ by no more than $2k$. In addition, we use a similar process to find the necessary condition for affinity groups of more than two elements. The average distance between any two elements in a g-element group is at most $2gk$. The same condition holds for affinity groups of g data sets.

The condition is necessary, as shown by the derivation. It is not sufficient. Using the case of two-element groups again, the bound $2k$ is for the worst case. In the best case, the difference between the average reuse distance is 0. An improvement is to check for the middle point, the bound of k. Another source of inaccuracy is that we do not check the reuse distance of each access but only the average distance calculated from the total distance. It is possible that the total distance is the same but individual reuses differ by more than k in distance. An improvement is to check the sum of each sub-set of memory accesses instead of the set of all accesses. The difficulty is to partition in the same way for accesses of the two data sets. We use the bins of the reuse signature and check the condition in each bin separately. Considering partition variations at the bin boundaries (as shown by the example in Figure 3), we apply the necessary condition to the sum of the average of all bins rather than the average of each bin. The improved condition is as follows.

Let the reuse signature have B bins after removing short-distance bins. Let X and Y be the two sets of data, and Avg_i^X and Avg_i^Y be the average reuse distance of the two data sets in the ith bin.

$$d = \sum_{i=1}^{B} |Avg_i^X - Avg_i^Y| \leq k \cdot B \quad (1)$$

The equation ensures that the average reuse distance per bin differs by no more than k. The left-hand side of the inequality is the difference between X and Y known as the Manhattan distance of the two vectors. For two-dimensional vectors, it is the driving distance of a taxi going from one place to another in downtown New York City (no relation to the driving time, however).

In addition, a reuse distance does not include the exact time of the data access. It is possible that two elements are accessed in the same reuse distance, but one in the first half of the execution, and the other in the second half. An improvement is to divide the execution trace into sub-parts and check the condition for each part.

The maximal difference between any two members of a g-element affinity group is no more than $2gk$. The condition is recursive because knowing group members requires knowing the group size first. The recursive equation can be solved iteratively. For each data set X, we find all other sets whose average distance differs no more than bk and let b range from 1 to $2g$. The solution is the largest b such that exactly $b-1$ data sets satisfy the condition. The process must terminate with the correct result.

In practice, we use a stricter condition to build a group incrementally. Initially each data set is a group. Then we traverse the groups and merge two groups if a member in one group and another member in the other group satisfy Equation 1. The process terminates if no more groups can be merged. We need to calculate the distance difference between any two data sets in $O(g^2)$ time. The iterative solutions takes at most $O(g^2)$. The incremental solution takes linear time if implemented using a work-list.

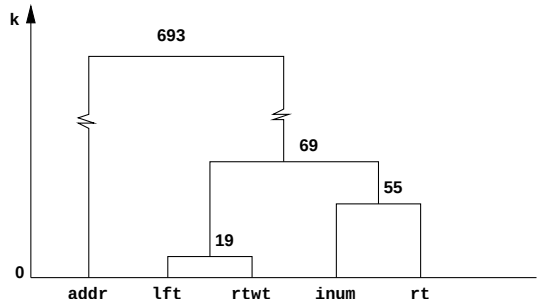

Figure 4: Dendrogram from k-distance analysis for the five tree node fields in *Cheetah*

The reference affinity forms a hierarchy for different k values. Figure 4 shows the hierarchy in a dendrogram. The reference affinity between *lft* and *rtwt* is strongest—their accesses are within a distance of 19 data elements. Other two fields, *inum* and *rt*, are used together within a distance of 55. These two groups are reused within $k = 69$. The last field, *addr*, has the least reference affinity, not accessed with other fields within hundreds of data.

3.3 Comparison with other methods

This section discusses other locality analysis and shows the three unique features of the k-distance analysis—the use of affinity groups, the comparison based on reuse signatures, and the use of the constant bound k. An experimental comparison follows in Section 5.

3.3.1 Compiler analysis

For programs written in regular loops and array access, a compiler can determine the exact access pattern and the best computation and data organization. However, for programs with complex control flows and indirect data access, a compiler must make conservative assumptions to ensure the benefit of a transformation. While profiling analysis is more generally applicable, results from

one profile may not reflect the access pattern in other executions. K-distance analysis alleviates this problem by relying on long reuse distances and the reuse signature, which often have consistent patterns across data inputs. In addition, the analysis considers the access pattern in multiple inputs (by merging reuse signatures) and further reduces the chance of a false positive. The negative conclusion from the analysis, i.e. two data do not have close reference affinity, is always true for the program as a whole: an execution violates the necessary condition in some program path.

3.3.2 *Frequency profiling*

Often in programs some data are more frequently used than others. Grouping the high frequency data often reduces the working set of a program. However, a question remains on how to place data that have the same access frequency, including the (larger) infrequently accessed data. Frequency is not the same as affinity. Data having the same frequency may not be accessed together at all. In comparison, the reference affinity gives a data partition based on the pattern of data reuse. It is important to measure distance by volume not time. An execution may nibble small data bits or devour huge data sets in the same amount of time. Reuse signature allows us to remove short-distance reuses as noises, regardless of their frequency or time distance.

An extension to frequency is the pair-wise affinity—the frequency that a pair of data are used together. The pair-wise affinity forms a complete graph where each datum is a node and the pair-wise frequency is the edge weight. However, the reference affinity is not transitive in a (pair-wise) graph. Consider the access sequence $abab..ab$ ... $bcbc..bc$: the pair-wise affinity exists for a and b, for b and c, but not for a and c. Hence the pair-wise affinity is indeed pair wise and cannot guarantee the affinity relation for data groups with more than two elements. Furthermore, the criterion for two data "accessed together" is based on preselected "cut-off" radii. In comparison, k-distance analysis defines affinity in data groups and measures the "togetherness" with a scale—the data volume between accesses.

3.3.3 *Statistical clustering*

Since the reuse signature is a vector, a natural impulse is to apply the sophisticated and readily available clustering techniques based on the well-grounded multivariate statistical theory. First proposed by MacQueen in 1967, k-means groups high-dimensional vectors by minimizing the within-group sum of distances to the group centroid [27]. It iteratively regroups points until a local minimum is reached [21]. It requires k, the number of groups, be part of the input. Affinity analysis, however, cannot pre-determine the number of affinity groups. An extension, x-means, finds the best k using the Bayesian Information Criterion (BIC). It compares different groups formed for different k's and chooses the one with the highest probability [32].

In our earlier study, k-means and x-means proposed many candidates, and a few showed good improvements [42]. However, we could not explain the results in any sensible way because little consistency existed between the best grouping of t clusters, $t+1$, and $t-1$ clusters. To us, the groups seemed to come out randomly, likely because we could not penetrate the two complex algorithms, each of which has a number of tunable parameters. For the same number of clusters, k-means produced entirely different groups than x-means did. The primary problem, as we learned, is that statistical clustering uses *relative closeness*. The grouping of any two points is determined, not by their position but by the position of other points. Figure 5 shows two vector spaces each containing three points. The points X and Y have a fixed position, but their grouping completely depends on the position of the third point Z. We can and we did alleviate the problem by introducing anchor points into the space, but we also realized that the right measure is the absolute closeness, not the relative closeness. For example, the benefit of grouping two data should be determined by how they are accessed, not by other data, which may not even be used at all in the same part of the execution. By using the absolute closeness, we simplified the problem and (happily) retired the somewhat unwieldy statistical tools.

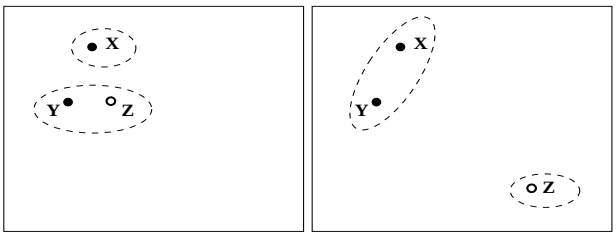

Figure 5: An example of statistical clustering. The points X and Y have a fixed position, but their grouping completely depends on the position of the third point Z.

3.3.4 *K%-distance*

K%-distance groups two reuse signatures X and Y (of length B) if the difference p, shown below, is less than k%. The difference in each bin, $|x_i - y_i|$, can be in the number of reuses, the sum of the reuse distance, or a combination of the two.

$$p_{<X,Y>} = \frac{\sum_{i=1}^{B} |x_i - y_i|}{\sum_{i=1}^{B} x_i + \sum_{i=1}^{B} y_i} \times 100\%$$

K%-distance is an improvement over statistical clustering. It measures the absolute closeness between reuse signatures. Compared to x-means or k-means, the result is tangible—the difference in each group is no more than k%. The partitions are hierarchical. The groups resulted from a lower k must be a finer partition of those from a higher k.

Despite being intuitive and hierarchical, k%-distance analysis has a problem. It does not really measure the absolute closeness. For example, 1% of one thousand is much smaller than 1% of one million. While the first case indicates close access, the latter does not, at least not to the same degree. We need a measure that is not relative to the length of reuse distances, the length of the execution, or the size of program data. The answer, as it turned out, is in our definition of the reference affinity: the link length k. It is a constant.

We describe k%-distance analysis as it was in our study—a halfway solution. By comparing k-distance and k%-distance, we show the importance of the constant bound, which plays the central role in both the affinity definition and the affinity testing. This connection is the most important discovery from this work. It unites the theory and the use of reference affinity.

4. DATA REORGANIZATION

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 access pattern of a program. A better way is to group

attributes based on their reference affinity. For arrays, the transformation is *array regrouping* [14, 15]. For structures, it is *structure splitting* [9, 8, 34]. This section describes the two transformations and their compiler support.

4.1 Array regrouping

Figure 6 shows an example. Initially the three coordinates of M molecules are stored in three arrays X, Y, and Z. They can be grouped into a single array whose first dimension has 3 elements, equivalent to an array of M structures. Fortran 90 and C allows grouping arrays of different types by using structures. Ding and Kennedy showed that array regrouping can be automatically applied in Fortran programs by a compiler [15]. We use the same compiler support in this work.

```
real*8 X(M), Y(M), Z(M)
```

(a) before grouping

```
real*8 XYZ(3,M)
```

(b) after grouping

Figure 6: Array regrouping example in Fortran 77

4.2 Structure splitting

To reduce the programming effort and ensure the correctness of the transformed programs, we have built a compiler that handles static type-safe C programs. The compiler support is just enough to evaluate reference affinity in experiments. It is not our purpose in this paper to develop a general technique for structure splitting or to argue that a general solution exists.

In static type-safe C programs, a compiler knows the type of every memory access. Given a program and a structure type targeted for splitting (we call it a *split type*), the compiler changes the allocation and the access of *all* objects of the split type (we call them *split objects*). It first pre-allocates space in field arrays. A split object is allocated from available entries in field arrays. A split object is no longer identified by a pointer but by an index. Figure 7 shows an example where the three fields of a structure type in Part (a) are split into two groups. Two field arrays are pre-allocated in Part (b) as arrays of structures. The pointers to a split object are changed to integer indices in the new program.

```
struct N {
    int val;
    struct N* left;
    struct N* right;
};
```

(a) before splitting

```
struct N_fragm0 {
    int val;
    unsigned int left;
};
struct N_fragm1 {
    unsigned int right;
};
struct  N_fragm0 f0_store[RESERVED];
struct  N_fragm1 f1_store[RESERVED];
```

(b) after splitting

Figure 7: Structure splitting example in C

As object pointers become array indices, object access becomes array access (and structure access if the array contains multiple fields). Through type analysis, the compiler knows all the places where a split object is accessed and makes conversion in those and only those places. Taking the pointer of a field in a split object is allowed, which returns a pointer to an element in a field array.

Three problems immediately arise with this scheme. First, a split object can be local to a function and should not permanently stay in field arrays. The compiler solves this problem by managing the upper-end of the field array as a stack and inserting allocation and free calls at the entry and the exit of the function. A field array is then shared by global and stack allocated objects in the same way as virtual memory is shared by heap and stack data, except that we manage a set of stacks for each split type.

The second problem happens when a split object is nested as a child inside a parent object. Three cases may happen. When the parent is split but the child is not, we support it by not splitting inside a field if it is a structure. When the parent and the child both split, we convert the child field into a pointer, which points to an independent object allocated when the parent object is allocated. We do not support the third case when the child is split but the parent is not.

Another major problem is the size of pre-allocation. The conversion of pointers to indices actually enables a solution through dynamic re-allocation. During an execution when the pre-allocated space is full, an allocator doubles the field arrays by allocating twice the space, copying the old array to the new space, changing the base of the field arrays, and reclaiming the space from the old array. The object access is unimpeded after the re-allocation. To support the array re-allocation, the compiler can no longer let pointers be taken from a field of a split object. In our test programs, pre-allocation is not an obstacle because the programmer specifies a bounded size for the main data structure, as done in all performance-critical programs. For example, *Cheetah* sets the maximal size of the splay tree by a compile-time constant. None of our programs needs re-allocation at run time. We note that our scheme ignores a host of issues such as union types, non-local jumps, exceptions, and concurrency because they do not arise in our test programs.

One type of splitting is no splitting, where the compiler does not change the layout of object fields, but it converts split objects from the pointer form to the array form. In the evaluation section, we will measure the effect of reference affinity using the array allocated version as the base case. We will also compare the (faster) performance of array version with that of the pointer version.

Chilimbi et al. first used structure splitting to improve data locality [9]. For type safe programs, they split an object into two parts, one storing frequently accessed fields and the other storing the rest. A pointer is inserted into the first part to link to the second part. Rabbah and Palem split structured data in C programs by allocating objects in large chunks where structure fields were stored in separate arrays [34]. Their method uses the address of the first field to identify the object and calculates the address of other fields at run time. It uses point-to analysis and dynamic checks to ensure correctness. It avoids the space and time cost of additional pointers as in the method of Chilimbi et al., but run-time (access) checks are needed for correctness even for static type-safe programs. In comparison, our array allocation relies on static type safety not run-time checking. It does not support as many types of programs or as efficient dynamic allocation as the other two methods do, but our method incurs least overhead when accessing the transformed data.

Previous work did not explore all choices of structure splitting. Chilimbi split structure fields into at most two parts [9]. Rabbah and Palem split structure fields completely [34]. Our compiler permits arbitrary structure splitting, which is needed for using and evaluating reference affinity.

5. EVALUATION

This section evaluates k-distance analysis and compares it with the alternative data layouts given by the programmer and four other methods described in Section 3.3.

5.1 Methodology

Test programs. Table 1 lists a set of 9 programs. The first four are array-based programs, including two from Spec95 suite and two dynamic programs originally from the Chaos group [12] for unstructured mesh and N-body simulation. The rest five programs use different tree structures including a splay tree for cache simulation, a quad-tree for image processing, and binary search trees for sorting and various other purposes. The cache simulator is part of the SimpleScalar tool set. The other four programs come from Olden benchmark set. We use different inputs for training and testing, as listed in the table. For a fair comparison, we only use one training input for each benchmark, despite that k-distance analysis can use multiple training runs.

We do not test more programs in part because our current compiler cannot safely split structures in all Olden programs, but also because the current set is statistically significant. All programs have at least 3 fields or arrays, five have 7 or more, and one program, *Swim*, has 14 arrays. A program with n arrays or structure fields has an exponential number (at least 2^{n-2} and $\sum_{i=0}^{n} i^{(n-i)}$ to be exact) of possible data layouts. The possible layouts is 4 for $n = 3$, 210 for $n = 7$, and over 6 million for $n = 14$. Therefore, the probability for a single method to consistently pick the best layout for *all* programs among *all* tested choices and on *all* tested machines is effectively zero, unless the method is indeed the best.

Platforms. The experiments use two machines, a 1.3GHz IBM Power4 processor with the AIX compiler, and a 2GHz Intel Pentium IV processor with the Linux *gcc* compiler. All testing programs are compiled at the optimization level *-O5 -qstrict* with AIX and *-O3* with Linux gcc respectively. The time is for complete executions on unloaded processors. We take the shortest time in 5 runs. The programs run 10% to 60% faster on IBM than on PC due to a faster processor and a better compiler. The only exception is *TSP*, which runs twice slower on IBM possibly because of the highly frequent floating-point square-root operations (used by *TSP* to calculate the Euclidean distance). We also tested the programs on a 250 MHz MIPS R10K processor on SGI Origin2000 using the MIPSpro compiler and a 336 MHz UltraSparc processor using the Sun compiler at the highest optimization level. The qualitative results are similar. The best layout on IBM is also the best on SGI or Sun. The programs run up to five times slower on SGI and many more times slower on Sun. The improvement is less dramatic because the memory problem is less severe on the two slower processors. We do not report SGI and Sun results for lack of space.

Tools. We use the source-level instrumentation and a run-time monitor to profile accesses for individual arrays and structure fields [16, 40]. They are accurate and efficient. For example in one execution of *Cheetah*, the monitor tracks the distinct access to 1.2 million data elements using a hashtable of less than 18 thousand entries. We use a 99%-accurate analyzer for reuse-distance analysis [17]. It measures reuse distance in long traces in effectively linear time and guarantees that the measured distance is between 99% and 100% of the actual distance. The two tools give the reuse signature for each array and structure field. The k-means and x-means tools come from Pelleg and Moore [32]. Compiler-based array regrouping is due to Ding and Kennedy [13, 15]. We implement k-distance, k%-distance, and frequency methods in MATLAB and array regrouping and structure splitting in our compilers as described in Section 4.

Analysis Time. The trace and affinity analysis takes a time proportional to the length of the execution and the number of data elements to be clustered. The profiling is more time consuming than the clustering. For all the tested programs, the profiling time is less than ten minutes. The k-distance analysis on reuse signatures takes less than one second.

5.2 Performance Comparison

We compare nine methods. The results are shown in Table 2 for the PC and Table 3 for the IBM machine. All execution times are in seconds. The last row is the arithmetic mean of the speedup. The first is the original layout coded by the programmer. The next two are k-distance for $k = 256$ and 64, followed by two k%-distance methods for 1% and 0.1%. Then it is x-means. We do not include k-means because it cannot pre-determine the number of affinity groups. The seventh method uses the access frequency to divide data into groups. It clusters the smallest set that accounts for at least 50% of all accesses into one group and stores the others in single arrays. The eighth method uses the compiler implementation from Ding and Kennedy [13, 15]. An interesting use of k-distance analysis is to find the worst data layout by reversing the reference affinity. It has no practical use other than showing the range of the effect from the data layout. The last column gives the execution time of data layout obtained by this *reverse k-distance*(*RK*) method. With the parameter 2048, it groups data whose average reuse distance is greater than $2K$. For all methods, the same data layout is tested on both machines.

The k-distance method for $k = 256$ should be the best method. It groups data that are almost always used within one to two kilobytes of data access, which fits comfortably in the L1 cache of all machines we use. For large cache, k should be greater. We stick to a single value because all other methods use a single parameter. Indeed, the analysis picks the best data layout for all programs on the two machines. *Swim* has 14 arrays and over 6 million possible choices, the analysis singles out a layout, which outperforms all others by a wide margin. The affinity hierarchy of *Swim* is a very impressive, large tree, as shown by Figure 8. On the PC, the layout from k-distance is 4% faster than the fastest alternative layout and 38% faster than the original layout. The improvement is smaller on IBM, 8% faster than the original. The margin is indisputable—no other known layout on both machines comes within 90% performance of this seemingly singular choice by the analysis.

For other programs, 256-distance improves *Tomcatv* and *Perimeter* by 25% and *TSP* by 20% on the PC. The average improvement is 12% on the PC and 4.5% on IBM. The improvement is more significant for programs that have many arrays or fields. The analysis does not blindly transform a program. The structure in *Bisort* given by the programmer has the best performance. The analysis recommends no change.

256-distance analysis runs slower in five places, *TreeAdd* against the original on the PC, *Tomcatv* against x-means on IBM, *Perimeter* against 64-distance and *Moldyn* against x-means and the reverse k-distance on the PC. However, the loss is extremely small (no more than 1% or 0.004 seconds) and happens on only one machine. For *TreeAdd* against the original, it loses by 0.7% on the PC but wins by 13% on IBM. Except for them, 256-distance always picks the

Table 1: Benchmark characteristics

Benchmark	Source	Description	Main data structure	Training input	Testing input
Swim	Spec95	shallow water equation	14 real arrays	$test(128^2)$	$ref(512^2)$
Tomcatv	Spec95	vectorized mesh generation	9 real arrays	$test(513^2)$	$ref(513^2)$
Mesh	Chaos Group	mesh structure simulation	7 float arrays	10k	10k
MolDyn	Chaos Group	molecular dynamics simulation	3 double arrays	13500 molecules	62500 molecules
Cheetah	SimpleScalar	fully associative LRU cache simulator	splay tree, 5 fields	jpeg encode 21.8K image	jpeg encode 940K image
Bisort	Olden	forward & backward integer sorting	binary tree, 3 fields	2^{17} nodes	2^{21} nodes
Perimeter	Olden	perimeters of regions in images	quad-tree, 7 fields	12 levels	12 levels
TreeAdd	Olden	recursive sum of values in a balanced-tree	binary tree, 3 fields	2^{18} nodes	2^{22} nodes
TSP	Olden	traveling salesman problem solver	binary tree, 7 fields	10^5 nodes	$4 * 10^6$ nodes

Table 2: Execution time (sec) on Intel Pentium IV

Benchmark	Orig	K=256	K=64	K=1%	K=0.1%	X-means	Freq=50%	Static	Worst
Swim	52.34	37.90	53.15	46.99	53.15	39.28	45.84	45.37	38.36
Tomcatv	45.37	36.43	36.43	36.43	36.43	37.65	37.35	36.85	38.15
MolDyn	69.78	69.78	69.78	69.78	69.78	69.68	69.78	69.78	69.55
Mesh	4.31	4.25	4.31	4.25	4.31	5.29	5.69	4.31	15.80
Cheetah	263.96	263.64	263.64	263.64	263.64	293.23	306.76	compiler	330.93
Bisort	12.16	12.16	12.16	12.16	12.16	14.38	14.22	analysis	15.98
Perimeter	0.035	0.028	0.026	0.028	0.028	0.028	0.029	not	0.039
TreeAdd	0.262	0.264	0.264	0.264	0.264	0.264	0.264	applicable	0.272
TSP	17.79	14.86	14.86	16.95	14.92	16.91	14.86		17.19
Average Speedup	**1.000**	**1.120**	**1.085**	**1.074**	**1.074**	**1.044**	**1.025**	**1.096**	**0.920**

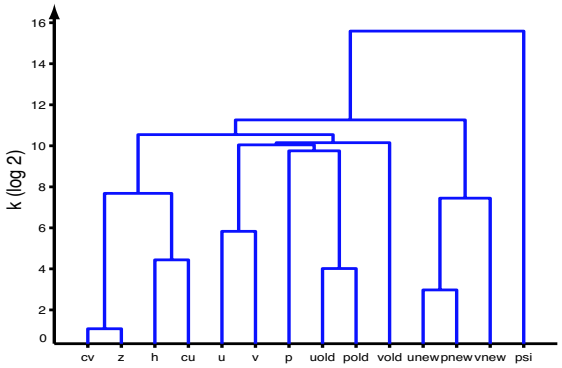

Figure 8: Dendrogram from k-distance for Swim

best data layout 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, among the 72 alternative layouts, 256-distances loses to only x-means on *MolDyn* by less than one tenth of a percent.

The layout by k-distance with $k = 64$ is overly conservative because it does not exploit the affinity at a distance greater than 64. It picks the same data layout as $k = 256$ for all but three programs, suggesting that the affinity groups have a short distance in the majority of the programs.

The two k%-clustering methods give competitive results, improving average performance by 7% on the PC and 3% on IBM. The two k values of 1% and 0.1% yield the same layout in all but three programs; $k = 1\%$ wins in two on the PC, but loses on two on IBM. This verifies that no single k% is the best, because k% does not have an absolute meaning and may be too small in one program but too large in another. In contrast, the k in k-distance has an absolute meaning—the volume distance.

X-means, using the relative closeness, performs worse than all methods using the absolute closeness. On average, the performance is improved by 4% on the PC but slightly impaired on IBM. X-means features bewilderingly different quality on the two machines. It gives the third best data layout for *Swim* on the PC, but the same layout is the third slowest on IBM. The three fields of *Bisort* have almost identical reuse signatures. Based on the relative closeness, x-means stubbornly splits them into two groups, losing performance by 17% on PC and 30% on IBM.

The frequency splitting using a single parameter performs worse on average than all other methods, showing that grouping on frequency is not as good as grouping on reuse signatures. The compiler analysis is conservative. It causes no slowdown except for *Swim* on IBM but has less benefit than k-distance analysis. It cannot yet analyze structure access in C programs.

Reference affinity has a very different effect on the two machines for the first two Fortran programs. The improvement for *Swim* is 8% on IBM but 38% on the PC. The reason is the compiler. The IBM compiler performs sophisticated loop transformations to improve the spatial locality, while the Gcc compiler does little, leaving amble room for improvement. The IBM compiler outperforms the static data regrouping of Ding and Kennedy for *Swim* but not for *Tomcatv* [15]. In both cases, reference affinity outperforms the best static compiler optimization.

On IBM, the reverse k-distance gives the slowest running time for all benchmarks except for *Perimeter*. The average loss is 12%. The results on the PC are mixed, likely for factors other than the

Table 3: Execution time (sec) on IBM Power 4

Benchmark	Orig	K=256	K=64	K=1%	K=0.1%	X-means	Freq=50%	Static	Worst
Swim	25.32	23.46	26.01	27.87	26.01	26.48	27.29	26.85	29.80
Tomcatv	21.74	20.70	20.70	20.70	20.70	20.60	23.74	20.90	23.80
MolDyn	52.22	52.22	52.22	52.22	52.22	52.25	52.22	52.22	52.79
Mesh	3.29	3.26	3.29	3.26	3.29	3.32	3.42	3.29	5.09
Cheetah	195.08	190.60	190.60	190.60	190.60	204.45	208.65	compiler	218.76
Bisort	8.07	8.07	8.07	8.07	8.07	10.44	10.25	analysis	11.16
Perimeter	0.025	0.021	0.022	0.021	0.021	0.021	0.024	not	0.024
TreeAdd	0.230	0.226	0.226	0.226	0.226	0.226	0.226	applicable	0.258
TSP	41.69	40.34	40.34	40.47	40.37	41.15	40.34		41.73
Average Speedup	**1.000**	**1.045**	**1.026**	**1.026**	**1.032**	**0.995**	**0.958**	**0.996**	**0.883**

data layout. However, it still gives the worst slowdown of 8% on average. By comparing the best (*k=256*) and the worst (*rk = 2048*) data layout, we observe that reference affinity affects performance differently, from almost no effect in *MolDyn* on both machines to 74% for *Mesh* on the PC and 36% for the same program on IBM. The average effect is 20% on the PC and 16% on the IBM machine.

Table 4 shows the effect of converting the five C programs from using pointer-based data to using array allocation. On average, the array version improves the original pointer version by 30% on the PC and 44% on the IBM machine. The only degration occurs for *TSP* on the PC. But k-distance splitting eventually improves the performance by 10%. The benefit of array allocation becomes more significant when a program deals with the inputs with a larger memory working set. An example is *Cheetah*. The input to the cache simulator is the access trace of the *ijpeg* encoding program from MediaBench. The original, pointer-based version outperforms the best array version when simulating the encoding of an image line by line with a memory footprint of thousands of kilobytes. When we change the encoding option to interlaced GIF files and enlarge the footprint to tens of megabytes, the encoding takes much longer, and the array version runs consistently faster than the pointer version. The *Cheetah* results in this section are for one of the larger inputs. We found a similar trend in other programs. The improvement from the reference affinity becomes greater when a program uses a larger input and takes longer to run. Next we quantify the improvement across all program inputs for one of our test programs.

Miss-rate improvement across all inputs. The effect of data transformation may change with program inputs. Our past work showed that *Tomcatv* had a predictable miss rate across all inputs [41]. Using that tool on a DEC Alpha machine, we draw the miss rate of 96KB cache (the L2 on-chip cache size size of Alpha) for the original and k-distance analysis with $k = 256$ as two curves in Figure 9. The data input is measured by the data size in cache blocks. Array regrouping reduces the miss rate by little to over 5% depending on the input. The vertical bar marks the data input used in our experiments. The difference is about 0.7% in the absolute miss rate. On the DEC machine, the corresponding speed improvement is 7.24%.

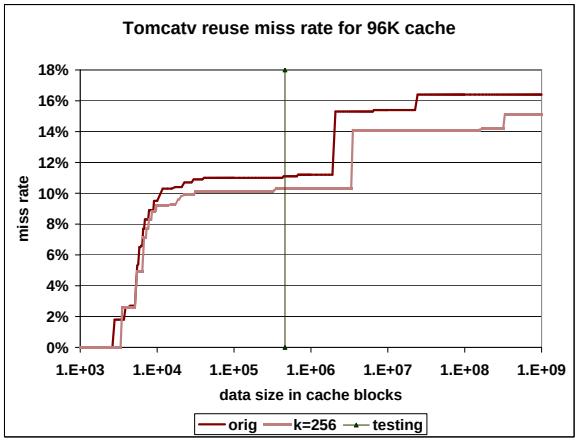

Figure 9: Reuse miss rate changes for *Tomcatv*

6. RELATED WORK

This section discusses mainly the past work on data transformation. The discussion is more bibliographical than technical. The reader should refer Section 3.3 for a technical comparison.

Early compiler analysis identifies groups of data that are used together in loop nests. Thabit used the concept of pair-wise affinity he called reference proximity [38]. Wolf and Lam [29] and McKinley et al. [39] used reference groups. Thabit showed that the optimal data placement using the pair-wise affinity is NP-hard [38]. Kennedy and Kremer gave a general model that considered, among others, run-time data transformation. They showed that the problem is also NP-hard [24]. Ding and Kennedy used the results of Thabit and of Kennedy and Kremer to prove the complexity of the partial and dynamic reference affinity [14]. To reduce false sharing in multi-treaded programs, Anderson et al. [4] and Eggers and Jeremiassen [22] grouped data accessed by the same thread. Anderson et al. optimized a program for computation as well as data locality, but they did not combine different arrays. Eggers and Jeremiassen combined multiple arrays for thread locality, but their scheme may hurt cache locality if not all thread data are used at the same time.

For improving the cache performance, Ding and Kennedy grouped arrays that are always used together in a program [14]. They gave the optimal array layout for strict affinity. They later grouped arrays at multiple granularity [15]. An earlier version of this work defined hierarchical reference affinity and tested two programs using x-means and k-means clustering [42].

Program profiling has long been used to measure the frequency of data access [25]. Seidl and Zorn grouped frequently accessed objects to improve virtual memory performance [35]. Using pair-wise affinity, Calder et al. [7] and Chilimbi et al. [10] developed algorithms for hierarchical data placement in dynamic memory al-

Table 4: Execution time(sec) comparison between array-based and pointer-based versions

Benchmark	Intel Pentium IV			IBM Power 4		
	Pointer-based	Array-based	K=256	Pointer-based	Array-based	K=256
Cheetah	290.78	263.96	263.64	209.39	195.08	190.60
Bisort	13.79	12.16	12.16	9.57	8.07	8.07
Perimeter	0.039	0.035	0.028	0.033	0.025	0.021
TreeAdd	0.584	0.262	0.264	0.598	0.230	0.226
TSP	16.32	17.79	14.86	42.10	41.69	40.34
Average Speedup	**1.000**	**1.300**	**1.388**	**1.000**	**1.438**	**1.509**

location. The locality model of Calder et al. was an extension of the temporal relation graph of Gloy and Smith, who considered reuse distance in estimating the affinity relation [19]. Chilimbi et al. split structure data in C and Java programs using pair-wise affinity [9]. Chilimbi later improved structure splitting using the frequency of data sub-streams called hot-streams [8]. Hot-streams combines dynamic affinity with frequency but does not yet give whole-program reference affinity.

Access frequency and pair-wise affinity do not distinguish the time or the distance of data reuses. Petrank and Rawitz formalized this observation and proved a harsh bound: with only frequency or pair-wise information, no algorithm can guarantee a static data layout within a factor of $k - 3$ from the optimal solution, where k is proportional to the size of cache [33].

Rabbah and Palem gave another method for structure splitting. It finds opportunities for complete splitting by calculating the *neighbor affinity probability* without constructing an explicit affinity graph [34]. The probability shows the quality of a given layout but does not suggest the best reorganization. Their splitting method is fully automatic as discussed in Section 4.

Reference affinity may change during a dynamic execution. Researchers have examined various methods for dynamic data reorganization [11, 13, 20, 30, 31, 37]. Ding and Kennedy found that consecutive packing (first-touch data ordering) best exploits reference affinity for programs with good temporal locality [13], an observation later confirmed by Mellor-Crummey et al. [30] and Strout et al [37]. Ding and Kennedy considered the time distance of data reuses and used the information in group packing. They also gave a unified model in which consecutive packing and group packing became special cases. In principle, the model of reference affinity can be used at run time to analyze sub-parts of an execution. However, it must be very efficient to be cost effective. On-line affinity analysis is a subject of our on-going study.

Matson et al. first used reuse distance to measure the performance of virtual memory systems [28]. The recent uses include those of Zhou et al. [43] and Jiang and Zhang [23], who studied file caching and showed a significant improvement for program, server, and database traces. At least five compiler groups have used reuse distance to analyze program locality [3, 5, 15, 26, 40]. Beyls and D'Hollander used per-reference distance pattern to annotate programs with cache hints and improved SPEC95 FP program performance by 7% on an Itanium processor [6]. Ding and Zhong analyzed large traces by reducing the analysis cost to near linear time. They found that reuse-distance histograms change in predictable patterns in many programs [17]. Zhong et al. used this result to predict cache miss rates across program inputs and cache configurations [41].

Loop transformations have long been used to arrange stride-one access to maximize spatial reuse (see examples in [1, 18, 29] or a comprehensive text [2]). Computation reodering is preferable because it makes no (negative) impact in other parts of a program. Still, this paper shows that using reference affinity, data transformation is beneficial for programs where loop transformations or other types of static techniques are inadequate because of the complex control flow and indirect data access.

7. CONCLUSIONS

Reference affinity gives a unique and hierarchical partition of program data. The reference affinity among the source-level data can be tested by k-distance analysis. The result of k-distance analysis has an intuitive meaning. The elements of an affinity group must be accessed within a volume distance of at most k. Experiments show that the new method uncovers rich affinity relations among the data in complex programs. When used in array and structure reorganization, k-distance analysis outperforms all other methods with remarkable consistency. The close agreement between theoretical properties and experimental observations suggests that reference affinity is an effective way to bridge the gap between the memory hierarchy of a machine and the linear data layout of a program.

8. ACKNOWLEDGMENTS

Ken Kennedy gave the name reference affinity [15]. The explanation shown in Figure 5 was due to Mitsu Ogihara. Kevin Stoodley pointed out the cause for the different effects of reference affinity on the IBM and the PC machines. The work and its presentation benefited from discussions with Trishul Chilimbi, Guang Gao, and our colleagues at University of Rochester and Rice University as well as the anonymous reviewers of the LCPC 2003 workshop and the PLDI 2004 conference. This work is supported by the National Science Foundation (Contract No. CCR-0238176, CCR-0219848, and EIA-0080124), the Department of Energy (Contract No. DE-FG02-02ER25525), and an equipment grant from IBM.

9. REFERENCES

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 2001.

[3] G. Almasi, C. Cascaval, and D. Padua. Calculating stack distances effi ciently. In *Proceedings of the fi rst ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, June 2002.

[4] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[5] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, August 2001.

[6] 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.

[7] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, Oct 1998.

[8] T. M. Chilimbi. Effi cient 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.

[9] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure defi nition. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[10] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[11] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured euler solver using software primitives. In *Proceedings of the 30th Aerospace Science Meeting*, Reno, Navada, January 1992.

[12] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientifi c computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.

[13] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[14] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proceedings of The 12th International Workshop on Languages and Compilers for Parallel Computing*, La Jolla, California, August 1999.

[15] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1), 2004.

[16] C. Ding and Y. Zhong. Compiler-directed run-time monitoring of program data access. In *Proceedings of the fi rst ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, June 2002.

[17] 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.

[18] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[19] N. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5), September 1999.

[20] H. Han and C. W. Tseng. Locality optimizations for adaptive irregular scientifi c codes. Technical report, Department of Computer Science, University of Maryland, College Park, 2000.

[21] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.

[22] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, July 1995.

[23] S. Jiang and X. Zhang. LIRS: an effi cient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Marina Del Rey, California, June 2002.

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

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

[26] Z. Li, J. Gu, and G. Lee. An evaluation of the potential benefi ts of register allocation for array references. In *Workshop on Interaction between Compilers and Computer Architectures in conjuction with the HPCA-2*, San Jose, California, February 1996.

[27] J. MacQueen. Some methods for classifi cation and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statisitics and Probability*, pages 281–297, 1967.

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

[29] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

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

[31] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affi ne array references. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, October 1999.

[32] D. Pelleg and A. Moore. X-means: Extending k-means with effi cient estimaiton of the number of clusters. In *Proceddings of the 17th International Conference on Machine Learning*, pages 727–734, San Francisco, CA, 2000.

[33] 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.

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

[35] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, Oct 1998.

[36] X. Shen, Y. Zhong, and C. Ding. Regression-based multi-model prediction of data reuse signature. In *Proceedings of the 4th Annual Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, November 2003.

[37] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

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

[39] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[40] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientifi c programs. In *Proceedings of Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington DC, March 2002.

[41] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate preidiction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.

[42] Y. Zhong, X. Shen, and C. Ding. A hierarchical model of reference affi nity. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, College Station, Texas, October 2003.

[43] Y. Zhou, P. M. Chen, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX Technical Conference*, June 2001.