A Hierarchical Model of Reference Affinity

Yutao Zhong, Xipeng Shen, and Chen Ding

Computer Science Department, University of Rochester
Rochester, NY 14627, USA
{ytzhong,xshen,cding}@cs.rochester.edu

Abstract. To improve performance, data reorganization needs locality models to identify groups of data that have reference affinity. Much past work is based on access frequency and does not consider accessing time directly. In this paper, we propose a new model of reference affinity. This model considers the distance between data accesses in addition to the frequency. Affinity groups defined by this model are consistent and have a hierarchical structure. The former property ensures the profitability of data packing, while the latter supports data packing for storage units of different sizes. We then present a statistical clustering method that identifies affinity groups among structure fields and data arrays by analyzing training runs of a program. When used by structure splitting and array regrouping, the new method improves the performance of two test programs by up to 31%. The new data layout is significantly better than that produced by the programmer or by static compiler analysis.

1 Introduction

The widespread use of hierarchical memory on today's PCs and workstations is based on the assumption that programs have locality. At the early days of virtual memory design, Denning defined *locality* as "a concept that a program favors a subset of its segments during extended intervals (phases)" and *locality set* as "the set of segments needed in a given program phase" [9]. Locality set measures the memory demand but does not suggest how to improve it. Abu-Sufah, working with Kuck, used data dependence information to estimate program locality and to reorder the program execution for better locality [1]. Thabit, working with Kennedy, analyzed the access affinity among data elements and used data placement to improve locality [32]. Subsequent research has examined a great number of locality models and their use in computation reordering, data reordering, or their combination.

In this paper we restrict our attention to locality models that are used in data transformation. Data placement improves memory performance by grouping useful data into the same or adjacent cache blocks or memory pages. On today's high-end machines from IBM, SUN, and companies using Intel Itanium and AMD processors, the largest cache in the hierarchy is composed of blocks of no smaller than 64 bytes. If only one four-byte integer is useful in each cache block, 94% of cache space would be occupied by useless data, and only 6% of cache is available for data reuse. A similar issue exists for memory pages, except that the utilization problem can be much worse. By grouping useful data together, data placement can significantly improve cache and memory utilization.

Data placement needs some model of reference affinity to tell which data are useful and should be grouped together. The past models are based on the access frequency. Thabit and many others used the frequency of data pairs called access affinity [32]. Chilimbi used the frequence of data "streams", which are subsequences of data access [4]. The frequency model does not consider time of the access. For example, suppose that a program executes in three phases that frequently access three data pairs x and y, y and z, and z and x respectively. If we use only frequency information, we may group all three elements into the same cache block, although they are never used together. The problem becomes worse in grouping for larger storage units such as a memory page because the chance of grouping unrelated data is much greater. In 1999, Ding and Kennedy used a model to find arrays that are always accessed together [11]. However, the compiler-based model does not address locality in programs with general data and complex control flow.

In this paper, we describe a new model of reference affinity. A set of data have *reference affinity* if they are always used together by a program. We say that they are in the same *affinity group*. This reference affinity model has two unique properties that are important for data placement. The first is consistency—the group of data elements are always accessed together. Placing group data in the same cache block always guarantees high space utilization. We will later define what we mean by "accessed together" and show how the consistency requirement can be relaxed to consider partial utilization of cache.

The second property is that the model has a hierarchical structure. The largest group is the set of all program data, if we treat the entire execution as one unit of time. As we change the granularity of time, we find a decomposition of program data into groups of smaller sizes until the extreme case when each data element is a group. Hierarchical groups allow us to fully utilize cache hierarchy. An affinity group used in cache-block packing needs at most a dozen elements, while a group used for a memory page may need over one thousand elements. These two properties distinguish this reference affinity model from other existing models especially frequency-based models.

The rest of this paper is organized as follows. We first define reference affinity and prove its consistency and hierarchical properties. We describe a new method for analyzing reference affinity at the source level and use it to improve cache utilization. This research is still in progress. We have not formulated all extensions of the basic concepts, nor have we evaluated our method on a broad class of programs or against alternative approaches. This is a preliminary report of our current findings.

2 Reference Affinity

This section first defines three preliminary concepts and gives two examples of our reference affinity model. Then it presents its formal definition and proves its properties including consistent affinity and hierarchical organization.

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 array index such as $T[a_x]$ to represent the logical time of an access a_x on trace T.

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 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 volume distance measures the volume of data accessed between two points of a trace. It is in contrast with the time distance, which is the difference between the logical time of two accesses. For example, the volume distance between the accesses to a and c in trace $abbbc$ is two because two distinct element is accessed in $abbb$. Given any three accesses in 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.

Mattson defined the volume distance between a pair of data reuses as LRU stack distance [23]. Volume distance can be measured in the same way as stack distance. Ding and Zhong recently gave a fast analysis technique that can measure volume distance in traces with tens of billions of memory accesses to hundreds millions of data [13]. We use Ding and Zhong's method in our experimental study, which will be presented in Section 4.

Based on the volume distance, we define a *linked path* on a trace. There is a linked path from a_x to a_y ($x \neq y$) if and only if there exist k accesses, $a_{x_1}, a_{x_2}, \ldots, a_{x_k}$, such that (1) $dis(a_x, a_{x_1}) \leq d \wedge dis(a_{x_1}, a_{x_2}) \leq d \wedge \ldots \wedge dis(a_{x_k}, a_y) \leq d$ and (2) $x_1, x_2, \ldots, x_k$, and x and y are all 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 d. We call d the *link length*. We will later restrict $x_1, x_2, \ldots, x_k$ 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 d for set S.

We now explain reference affinity with two example address traces in Fig. 1. The "..." represents accesses to other data elements other than w, x, y, and z. In the first example, accesses to x, y, and z are in three time ranges. They have consistent affinity because they are always accessed together. They belong to the same affinity group. The consistency is important for data placement. For example, x and w are not always used together, then 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 different frequency, and mixed with accesses to other data. However, one property holds in all three time ranges—the accesses to the three elements are connected by a linked path with a link length of at most 2.

As we will prove later, affinity groups are parameterized by the link length d and for each d, they form a partition of program data. The second example in Fig. 1 shows that group partition has a hierarchical structure for different link lengths. 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 structure is hierarchical with respect to the link length: groups at a smaller link length are subsets of groups at a greater link length. The hierarchical structure is useful in data placement because it may find different-sized affinity groups that match the capacity of the multi-level cache hierarchy.

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

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

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

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

Fig. 1. Examples of reference affi nity model and its properties

We now present the formal definition of 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 reference affinity) with the link length d 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 $x_1, x_2, \ldots, x_k \in G$ such that $dis(a_x, a_{x_1}) \leq d \wedge dis(a_{x_1}, a_{x_2}) \leq d \wedge \ldots \wedge dis(a_{x_k}, a_y) \leq d$*
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 d, the affinity groups defined by Definition 1 form a unique partition of program data.*

Proof. We show that any element x of program data belongs to one and only one affinity group at a link length d. 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 $y_1, \ldots, y_k \in G_1$ and an access a_x such that $dis(a_y, a_{y_1}) \leq d \wedge \ldots \wedge dis(a_{y_k}, a_x) \leq d$. 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 $dis(a_x, a_{z_1}) \leq d \wedge \ldots \wedge dis(a_{z_m}, a_z) \leq d$.

If $y_1, \ldots, y_k \notin \{z_1, \ldots, z_m\}$, 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 $y_i = z_t$. Then we have a linked path from a_y to a_{y_i}. Since $y_i = z_t \in G_2$, 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 $dis(a_y, a_{y_1}) \leq d \wedge \ldots \wedge dis(a_{y_{i_1}}, a_{y_i}) \leq d \wedge dis(a_{y_i}, a_{z'_1}) \wedge \ldots \wedge dis(a'_{z_n}, a_z) \leq d$. Now y_i belongs to $G_1 \cap G_2$, just like x. We

have come back to the same situation except 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 for ever because each step shortens the path from y to the next chosen access 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 hold 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 types 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 seen, 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. 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 d. The uniqueness removes any possible conflict, which would happen if a data element could appear in multiple affinity groups.

Next we prove that 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 d and d' ($d < d'$), the affinity groups at d form a finer partition of affinity groups at d'.*

Proof. We show that any affinity group at d is a subset of some affinity group at d'. Let G be an affinity group at d and G' be the affinity group at d' that overlaps with G ($G \cap G' \neq \emptyset$). Since any $x, y \in G$ are connected by a linked path with link length d, they are connected by a linked path with a larger link length d'. According to the proof of Theorem 1, $G \cup G'$ is an affinity group at d'. 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 is always accessed together. When one element is accessed, all other elements will be accessed within a time range with a bounded volume distance.

Theorem 3 *Given an address trace with an affinity group G at link length d, any time an element x of G is accessed at a_x, there exists a time range that includes a_x and at least one access to all other members of G, and the volume distance of the time range is no greater than $2d|G| + 1$, 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 $a_{x_1}, a_{x_2}, \ldots, a_{x_k}$. The volume distance from a_w to a_x is $dis(a_w, a_x)$. It is no greater than $dis(a_w, a_{x_1}) +$

$dis(a_{x_1}, a_{x_2}) + \ldots + dis(a_{x_k}, a_x)$, which is $(k+1)d \leq |G|d$. The bound of the volume distance from a_x to a_v is the same. Considering that a_v needs to be included in the time range, the total volume distance is at most $2d|G| + 1$.

The strict affinity requires that members of an affinity group are always accessed together. In many cases, a group of data may often be accessed together but not always. We can relaxed the first condition to require a group member to be accessed with other members k% of the time instead of all the time. The formal definition is below. The only change is the first condition.

Definition 2 Partial Reference Affinity *Given an address trace, a set G of data elements is a partial ($k\%$) affinity group with the link length d if and only if*

1. *for any $x \in G$, at least k% accesses a_x has a linked path from a_x to some a_y for each other member y in G*
2. *adding any other element to G will make Condition (1) impossible to hold*

Partial affinity groups do not produce unique partition of program data. The structure is not strictly hierarchical. The loss in consistency and organization depends on k. As on-going work, we are currently quantifying the bound of this loss as a function of k.

3 Clustering Analysis

The purpose of clustering analysis is to identify affinity groups. Data elements that tend to be accessed simultaneously should be clustered into the same group. We use k-means and its extension, x-means, to statistically measure the similarity of the reuse behavior of individual data elements and do the clustering.

K-means is a popular statistical clustering algorithm. It was first proposed by MacQueen [22] in 1967. The optimization criterion implied in k-means is *sum-of-squares criterion* [16]. The aim is to minimize the total within-group sum of squares. The basic idea of the algorithm is an iterative regrouping of objects until a local minimum is reached[18]. A sketch of the algorithm is as the follows:

1. Initialize with arbitrarily selected *k* centroids for *k* groups;
2. Assign each object to the closest centroids;
3. For each group, adjust the centroid to be the point denoted by the means of all objects assigned to that group;
4. If there are any changes in step 2 or 3, go to step 2; otherwise, stop.

One problem of k-means is that the value of *k* need to be specified at the beginning. For our affinity analysis, we may not know the optimal number of groups at the first place. Therefore, we also apply the extension of k-means, *x-means* [27] in our analysis. X-means relies on BIC (Bayesian Information Criterion) to compare different clusterings formed for different *k*'s. Based on BIC calculation, it approximately measures the

probability of each clustering given the original data set. Then the one with the highest probability is chosen.

According to the definition given in Section 2, an accurate way to identify affinity groups would be recording and comparing the reference trace of each data element. However, the time and space overhead of this approach would be high. Therefore, we propose an approximate estimation of affinity groups according to the reuse distance distribution of individual data elements. Reuse distance is equivalent to volume distance between two adjacent reuses to the same datum. We use the efficient measurement described in [13] to collect reuse distance information. For an array, we do not distinguish references to different array elements but view the whole array as a single data element. For a structure, we consider the accumulated reuse distance distributions of the accesses to each structure field of all instances of the structure. For example, a tree structure composed of two fields *left* and *right* will be considered two data elements. No matter how many objects of this structure type will be dynamically allocated, the references to the first field of these objects will always be counted as the accesses to the first data element. The same rule is applied to the references to the second field of allocated objects.

For any datum, the whole reuse distance scope for one execution is from zero to the maximal reuse distance that occurs in the execution. We divide this scope into a set of ranges, each in the form of $[d_1, d_2)$, where d_1, d_2 are two reuse distance values $(d_1 \leq d_2)$. Then for each range, we count the number of references whose reuse distance falls into that range and calculate the average distance of these reuses. The set composed of the number of references within each range forms a counter vector. The set of the average distance calculated for each range forms a distance vector. These two vectors are collected for every data element that we target to be grouped. They each describes the reuse behavior of the corresponding data element by locating them in an N dimensional space, while N is the number of reuse distance ranges considered. Since references with a long reuse distance have more significant effect on performance, we emphasize on such references in clustering analysis. For all experiments reported in this paper, only references with a reuse distance longer than 2048 are used in clustering and the reuse distance ranges are divided linearly with a constant length of 2048. In another word, the reuse distance ranges we considered in clustering analysis begin with $[2048, 4096)$ and go on with $[4096, 6144)$, $[6144, 8192)$, ..., and so forth.

An example with the above setting is given in Table 1. Suppose there are 10 reuses to the $left$ field of the instances of a tree structure. Their reuse distances are (sorted incrementally): $\{2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000\}$. We will construct 3 reuse distance ranges for this field and Table 1 describes the statistics collected for each range. The last two columns of the table list the two vectors to be clustered as $(4, 4, 2)$ and $(3250, 5250, 6750)$ respectively.

Our overall algorithm is shown in Fig. 2.

4 Evaluation

In this section, we measure the effect of the affinity group clustering by reorganizing data layout according to clustering results and comparing performance changes.

Table 1. Example of statistics collected for each data element and used in clustering

Range No.	Range scope	Reuse distance set	No. reuses	Avg. reuse distance
1	$[2048, 4096)$	$\{2500, 3000, 3500, 4000\}$	4	3250
2	$[4096, 6144)$	$\{4500, 5000, 5500, 6000\}$	4	5250
3	$[6144, 8192)$	$\{6500, 7000\}$	2	6750

```
algorithm ClusteringAnalysis
  input: V_Data: reuse distance distribution vectors for each data elements
  output: optimal data grouping
  begin
     Apply x-means on V_Data, K = No. of clusters recommended by x-means;
     for (i = K-1 ; i<K+1;i++)
        Apply k-means on V_Data with i as the specifi ed No. of clusters;
     end for
     Try training runs with all groupings found by x-means and k-means;
     Compare performance of training runs and fi nd the optimal data grouping;
  end
end algorithm
```

Fig. 2. Clustering analysis for data grouping

Two Test Programs We test on two different programs: $Cheetah$ and $Swim$. $Cheetah$ is a cache simulator written in C included in SimpleScalar suite 3.0. It constructs a splay tree to store and maintain the cache content. The tree structure of $Cheetah$ is composed of seven fields. In our experiments, we check the reuse behavior of each individual field within the tree structure and cluster them into groups. According to different clustering, we implement different versions of structure splitting on the source file and compare their performance. $Swim$ from Spec95 calculates finite difference approximations for shallow water equation. The reuse distance distributions of the fourteen arrays of real type in this Fortran program are collected and used in clustering. Then, the source file is changed by merging arrays clustered in the same group into a single array. Again, the performance of different versions are compared. These experiments also explore the potential uses and benefits of data clustering based on locality behaviors.

Clustering Methods We use the k-means and x-means analyzing tool implemented by Pelleg and Moore at Carnegie Mellon University [27]. Each data object to be clustered is represented by a set of feature values, each collected for a given reuse distance range. Two types of feature are considered: the number of reuses and the average reuse distance. The reuse distance ranges have a fixed length of 2048, as described in Section 3.

Platforms The experiments are performed on three different machines, including a 250MHz MIPS R10K processor with SGI MISPro compiler, a Sun Sparc U4500/336 processor and a 2GHz Intel PentiumIV processor with Linux gcc compiler. All programs are compiled with optimization flag *-n32 -Ofast* or *-O3* respectively.

Structure Splitting Table 2 lists the clustering results for the tree structure in $Cheetah$. The input to $Cheetah$ simulator is an access trace from *JPEG* encoding images sizing from dozens to tens of thousands bytes.

Table 2. Clustering for tree structure of $Cheetah$. K-means gives results for 2 to 4 clusters, we show only 2 and 3

Input image size	Clustering data	Clustering method	No. clusters	Clustering
43 bytes	No. reuses	k-means	3	(addr) (inum rt) (lft rtwt)
			2	(addr) (inum lft rt rtwt)
		x-means	3	(addr inum rt) (lft) (rtwt)
	Avg. reuse distance	k-means	3	(addr) (inum lft rtwt) (rt)
			2	(addr) (inum lft rt rtwt)
		x-means	3	(addr) (inum lft rtwt) (rt)
2.58K bytes	No. reuses	k-means	3	(addr) (inum lft rtwt) (rt)
			2	(addr) (inum lft rt rtwt)
		x-means	3	(addr) (inum lft rtwt) (rt)
	Avg. reuse distance	k-means	3	(addr) (inum) (lft rt rtwt)
			2	(addr inum) (lft rt rtwt)
		x-means	3	(addr) (inum) (lft rt rtwt)
21.8K bytes	No. reuses	k-means	3	(addr) (inum lft rtwt) (rt)
			2	(addr) (inum lft rt rtwt)
		x-means	3	(addr) (inum lft rtwt) (rt)
	Avg. reuse distance	k-means	3	(addr) (inum lft rtwt) (rt)
			2	(addr) (inum lft rt rtwt)
		x-means	3	(addr) (inum lft rtwt) (rt)

Although the tree structure consists of seven fields: *rtwt*, *rt*, *lft*, *inum*, *addr*, *grpno* and *prty*, the table only list the first five of them. The reason is the simulator for fully associative LRU cache only accesses the first five fields and we apply clustering only on them. Clustering the other two is trivial. The first column of the table gives the size of the encoded image. Column 2 and 3 describe how the the clustering is applied. The fourth column contains the number of clusters identified while the last column lists the clustering result. While k-means gives results for two to four clusters, we only show groupings with two and three clusters here.

The clustering results shown in Table 2 have two important features. First, the clustering on the five fields varies across different inputs or different clustering algorithms. Second, although there is no single winner, the clustering indicates a strong affinity among *rtwt*, *rt* , *lft* and *inum*. Therefore, we choose to reorganize the tree structure by grouping these four fields in different ways. Table 3 lists the structure splittings we tested.

Each row of Table 3 describes a grouping of the seven fields of the tree structure. Version $orig$ is an array-based version of the original $Cheetah$. In $orig$, the tree nodes are contained in a big pre-allocated array instead of dynamically allocated at run time. This array-based $Cheetah$ simulator is faster than the original $Cheetah$ (over 10%

Table 3. Different structure splitting for $Cheetah$

Version No.	Grouping
$orig$	(addr inum grpno lft prty rt rtwt)
$v1$	(addr) (inum) (grpno) (lft) (prty) (rt) (rtwt)
$v2$	(addr) (inum) (grpno) (lft rtwt) (prty) (rt)
$v3$	(addr) (inum) (grpno) (lft) (prty) (rt rtwt)
$v4$	(addr) (inum lft rt rtwt) (grpno) (prty)

faster when tested on an SGI processor). All other versions modify version $orig$ to implement structure splitting. $V1$ divides the structure into seven groups, each containing a single field. The other three versions group in different ways according to the similarity measured by the clustering analysis. We change the source files by hand to get different versions. The access trace of $JPEG$ encoding a testing image size of 272 KB is used as the testing input. Different versions of $Cheetah$ are compiled and run on the three platforms described above and the running times are collected by the standard $time$ utility. Table 4 summaries the experiment results.

Table 4. Performance of different structure splittings for $Cheetah$

Version No.	2GHz Intel PentiumIV	250MHz MIPS R10K	336MHz UltraSparc II
$orig$	23.4s	59.7s	93.7s
$v1$	23.7s	71.9s	91.7s
$v2$	20.7s	59.2s	91.2s
$v3$	20.1s	58.2s	92.5s
$v4$	19.3s	56.0s	89.1s
Best improvement over orig(%)	17.5	6.20	4.93
Best improvement over v1(%)	18.6	22.1	2.84

For each version listed, Table 4 gives the execution time on the three platforms. The user time reported by $time$ command is used as the execution time. The comparison between the first and second rows of the table shows there is no clear benefit by simply dividing the original structure into individual fields. Version $v1$ runs slower than $orig$ on both MIPS and Pentium machines. However, by grouping the fields with similar reuse behavior together, there is almost always a performance gain from other versions. Version $v4$ is consistently the best among all the groupings. It is up to 17.5% faster than the original version. This shows the clustering analysis based on reuse distance distribution is effective in identifying affinity groups.

Array Grouping $Swim$ has fourteen arrays with the same size. We apply clustering analysis on these arrays and merge arrays in the same cluster. Table 5 describes a subset

of the clustering results. We tested the performance of these groupings. The training data for clustering analysis was collected by running $Swim$ with an input matrix of size 32×32.

Table 5. Clustering for arrays of $Swim$. K-means gives results for 2 to 13 clusters, we show only 7 and 8

Clustering data	Clustering method	No. of clusters	Clustering	Version No.
No. of reuses	k-means	8	(cu h)(cv z)(p u v)(pold uold)(pnew unew)(psi)(vnew)(vold)	v1
		7	(cu cv h z)(p)(pnew unew vnew)(pold uold vold)(psi)(u)(v)	v2
	x-means	8	(cu cv h z)(p u v)(pnew)(pold uold)(psi)(unew)(vnew)(vold)	v3
Avg. reuse distance	k-means	7	(cu cv h z)(p)(pnew unew vnew)(pold uold vold)(psi)(u)(v)	v2
	x-means	8	(cu cv h z)(p)(pnew unew vnew)(pold uold)(psi)(u)(v)(vold)	v4
static analysis			(cu cv)(h z)(p)(pnew unew vnew)(pold uold vold)(psi)(u v)	static

The last row of Table 5 includes an array grouping based on static analysis. It was obtained by compiler analysis [11]. One restriction of array merging is the original arrays must have exactly the same size in all dimensions. This can be checked manually or by a compiler. The transformation process to get different grouping versions at source-level is semi-automatic. We tested $Swim$ for an input matrix of size 512×512. Table 6 gives the execution time for the original $Swim$ and all the five grouping versions.

Table 6. Performance of different array groupings for $Swim$

Version No.	2GHz Intel PentiumIV	250MHz MIPS R10K	336MHz UltraSparc II
$orig$	72.7s	156.7s	268.1s
$static$	57.5s	155.3s	239.6s
$v1$	59.8s	153.9s	232.5s
$v2$	50.1s	156.0s	226.7s
$v3$	54.1s	155.3s	221.4s
$v4$	50.1s	145.8s	226.7s
Best improvement over orig(%)	31.1	6.96	22.5
Best improvement over static (%)	12.9	6.12	13.2

Table 6 shows that the array groupings identified by clustering analysis outperforms the grouping based on static analysis most of the time. Version $v4$, identified as the optimal clustering by x-means method, is the best one on all machines. It reduces the execution time by up to 31.1% compared to the original version and up to 13.2% compared to the static analysis version.

5 Related Work

An effective method for fully utilizing cache is to make data access contiguous. Instead of rearranging data, the early studies reordered loops so that the innermost loop traverses data contiguously within each array. Various loop permutation schemes were studied for perfect loop nests or loops that can be made perfect, including those by Abu-Sufah et al. [2], Gannon et al. [15], Wolf and Lam [33], and Ferrante et al. [14]. McKinley et al. developed an effective heuristic that permutes loops into memory order for both perfect or non-perfect nested loops [24]. Loop reordering, however, cannot always achieve contiguous data traversal because of data dependences. This observation led Cierniak and Li to combine data transformation with loop reordering [6]. Kremer developed a general formulation for finding the optimal data layout that is either static or dynamic for a program at the expense of being an NP-hard problem and showed that it is practical to use integer programming to find an optimal solution [20]. Computation reordering is powerful when applicable. However, in many programs, not all data accesses in all programs can be made contiguous.

Alternatively, we can pack data that are used together. Early studies used the frequency of data access, measured by sample- and counter-based profiling by Knuth [21] and static probability analysis by Cocke and Kennedy [7] and by Sarkar [29]. Frequency information is frequently used in data placement, as we reviewed in the introduction. In addition, Chilimbi et al. split Java classes based on the access frequency of class members [5]. In addition to packing for cache blocks, Seidel and Zorn packed dynamically allocated data in memory pages [30]. Access frequency does not distinguish the time of access: that a pair or a group of data are frequently accessed does not mean that they are frequently accessed together, and that a group of data are accessed together more often than other data does not mean the data group are accessed together always or most of the time. In a recent paper, Petrank and Rawitz formalized this observation and proved a harsh bound: with only pair-wise information, no algorithm can find a static data layout that is always within a factor of $k-3$ from the optimal solution, where k is proportional to the size of cache [28]. Unlike reference affinity, the frequency-based models do not find data groups that are always accessed together. Neither do they partition data in a hierarchy based on their access pattern.

Eggers and Jeremiassen grouped data fields that were accessed by a parallel thread to reduce false sharing [19]. Ding and Kennedy regrouped Fortran arrays that are always accessed together to improve cache utilization [11]. Ding and Kennedy later extended it to group high-dimensional data at multiple granularity [12]. While the previous studies used compiler analysis, this work generalizes the concept of reference affinity to address traces. It also proposes a new profiling-based method for finding affinity groups among source-level data. Preliminary results show that the new method out-performs the data layout given by either the compiler or the programmer.

The above data packing methods are static and therefore cannot fully optimize dynamic programs whose data access pattern changes during execution. Dynamic data placement was first studied under an inspector-executor framework [8]. Al-Furaih and Ranka examined graph-based clustering of irregular data for cache[3]. Other models include consecutive packing by Ding and Kennedy [10], space-filling curve by Mellor-Crummey et al. [25], graph partitioning by Han and Tseng [17], and bucket sorting by

Michell et al [26]. Several studies found that consecutive packing compared favorably with other models [25, 31].

6 Summary

We have defined a new reference affinity model and proved its three basic properties: consistent groups, hierarchical organization, and bounded reference distance. We have described a clustering method to identify affinity groups among source-level structure fields and data arrays. The method uses data reuse statistics collected from training runs. It uses k-means and x-means clustering algorithms as a sub-procedure and explores a smaller number of choices before determining the reference affinity. When used by structure splitting and array grouping, the new method reduces execution time by up to 31%. It outperforms previous compiler analysis by up to 13%. As on-going work, we are formulating partial reference affinity, studying more accurate ways of reference affinity analysis, and exploring other uses of this locality model in program optimization.

7 Acknowledgment

This work is supported by the National Science Foundation (Contract No. CCR-0238176, CCR-0219848, and EIA-0080124) and the Department of Energy (Contract No. DE-FG02-02ER25525). We would like to thank Dan Pelleg and Andrew Moore for their assistance with the k-means and x-means toolkit.

References

1. W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.
2. 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.
3. I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of International Parallel Prcessing Symposium and Symposium on Parallel and Distributed Processing*, Orlando, Florida, April 1998.
4. 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.
5. T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure defi nition. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.
6. M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, 1995.
7. J. Cocke and K. Kennedy. Profi tability computations on program fbw graphs. Technical Report RC 5123, IBM, 1974.
8. 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.

9. P. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), January 1980.
10. 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.
11. 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.
12. C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings of International Parallel and Distributed Processing Symposium*, San Francisco, CA, 2001. http://www.ipdps.org.
13. 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.
14. J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, Aug. 1991. Springer-Verlag.
15. D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct. 1988.
16. A. D. Gordon. *Classifi cation*. Chapman and Hall, 1981.
17. 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.
18. J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.
19. 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.
20. K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4), 1998.
21. D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.
22. 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.
23. R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
24. 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.
25. J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming*, 29(3), June 2001.
26. 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.
27. 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.

28. 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.
29. V. Sarkar. Determining average program execution times and their variance. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, Oregon, January 1989.
30. 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.
31. 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.
32. K. O. Thabit. *Cache Management by the Compiler*. PhD thesis, Dept. of Computer Science, Rice University, 1981.
33. 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.