

A Hierarchical Model of Data Locality*

Chengliang Zhang Chen Ding
Mitsunori Ogihara

Computer Science Department
University of Rochester
734 Computer Studies Bldg
Rochester, NY 14627

{zhangchl,cding,ogihara}@cs.rochester.edu

Yutao Zhong

Computer Science Department
George Mason University
MSN 4A5
4400 University Drive
Fairfax, VA 22030
yzhong@cs.gmu.edu

Youfeng Wu

Programming Systems Research Lab
Intel labs
2200 Mission College Blvd
Santa Clara, CA 95052
Youfeng.wu@intel.com

Abstract

In POPL 2002, Petrank and Rawitz showed a universal result—finding optimal data placement is not only NP-hard but also impossible to approximate within a constant factor if $P \neq NP$. Here we study a recently published concept called *reference affinity*, which characterizes a group of data that are always accessed together in computation. On the theoretical side, we give the complexity for finding reference affinity in program traces, using a novel reduction that converts the notion of distance into satisfiability. We also prove that reference affinity automatically captures the hierarchical locality in divide-and-conquer computations including matrix solvers and N-body simulation. The proof establishes formal links between computation patterns in time and locality relations in space.

On the practical side, we show that efficient heuristics exist. In particular, we present a sampling method and show that it is more effective than the previously published technique, especially for data that are often but not always accessed together. We show the effect on generated and real traces. These theoretical and empirical results demonstrate that effective data placement is still attainable in general-purpose programs because common (albeit not all) locality patterns can be precisely modeled and efficiently analyzed.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—optimization, compilers

General Terms Theory, Algorithms, Performance

Keywords Hierarchical data placement, program locality, reference affinity, volume distance, NP-complete, N-body simulation

1. Introduction

Data placement becomes increasingly important to programming language design as programmers routinely improve performance or energy efficiency by better utilizing the cache memory in processors, disks, and networks. Different memory levels come with different sizes and configurations. The hardware configuration, how-

* This research has been supported by DOE (DE-FG02-02ER25525) and NSF (CCR-0238176)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

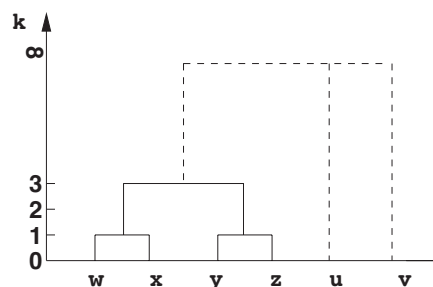
POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.
Copyright © 2006 ACM 1-59593-027-2/06/0001...\$5.00.

ever, may not be fully visible to a user. In addition, a program may run on machines with different configurations. As the programming for specific memory levels becomes increasingly untenable, solutions for *hierarchical data placement* are developed in separate application domains including matrix solvers [16], wavelet transform [7], N-body simulation [30], and search trees [3], where the program data are recursively decomposed into smaller blocks. By exploiting the inherent locality in computation, hierarchical data placement optimizes for any and all cache sizes. While most studies examined specific computation tasks, in this work we show that a general model can be used to derive the hierarchical data placement from program traces without user's knowledge of the meaning of the program.

While the data placement is sensitive to a machine, it is first and foremost driven by the computation order. In fact, any layout is perfect if the program traverses the data contiguously. Given an arbitrary data access trace, we say a group of data have reference affinity if they are always accessed together in the trace [44]. The closeness is parameterized by the *volume distance* (denoted by k), which is the volume of data between two accesses in a trace. We also call it the *reuse distance* if the two accesses are to the same datum. Changing k , reference affinity gives a *hierarchical* partition of program data. We show an example here and give the formal definitions in the next section. Figure 1 (a) shows a trace, where different letters represent accesses to different data, and “...” means accesses to data other than those shown in the trace.

w x w x u y z . . . z y z v x w x w . . .

(a) Example data access sequence over time



(b) The reference affinity gives a hierarchical relation in data

Figure 1. An example reference affinity hierarchy

Reference affinity gives a hierarchical relation shown as a dendrogram in Figure 1(b). The top of the hierarchy ($k = \infty$) is the set of all data $\{u, v, w, x, y, z\}$, which have the weakest affinity. The group $\{w, x, y, z\}$ have stronger affinity than they do with u and v (when $k = 3$). Inside this group, $\{w, x\}$ have closer affinity ($k = 2$), so do $\{y, z\}$. At the bottom of the hierarchy ($k = 0$), each data element becomes an affinity group. The affinity hierarchy enables the hierarchical data placement, which is simply the order (or its reverse) of the leaves in the dendrogram. The hierarchical placement improves the spatial locality for all cache configurations. When a data element is loaded, the following computation accesses more likely the neighboring data than the distant data. As shown by this example, reference affinity converts a computation trace to a hierarchical data layout.

In this paper we first present two theoretical results on reference affinity. The first is the complexity of finding reference affinity. We give polynomial-time algorithms for cases $k = 1$ and $k = 2$. We prove that the problems are either NPC or NP-hard when $k \geq 3$. Second, we prove that reference affinity automatically captures the hierarchical locality in divide-and-conquer type computations including blocked matrix solvers and N-body simulation. The proof holds even when data are divided into non-uniform sections and traversed in any order.

Despite of the theoretical complexity, efficient heuristics exist. We present a new analysis method based on sampling. We show through experiments that the new technique is more accurate than the previously published approximation method [44], especially for partial reference affinity where a group of data is often but not always accessed together. We show two new uses of reference affinity. The first is finding hierarchical data layout in recursive matrix multiplication, and the second is improving the code layout of seven SPEC 2000 applications.

The volume distance has been difficult for theoretical analysis because data may appear in different orders with different frequencies while still yielding the same volume distance. It raises interesting problems different from those in traditional graph and streaming domains. In this work, we present two new proof techniques that link between the volume distance of memory references and the affinity of data groups. The first contains a reduction that converts the problem of data volume into formal logic. The second contains a construction that connects the recursive structure of computation and the hierarchical relation of data.

In POPL 2002, Petrank and Rawitz showed a universal result—finding optimal data placement is not only NP-hard but also impossible to approximate within a constant factor if $P \neq NP$. This work shows a finer partition. On the one hand, good data placement is possible because reference affinity exists in most programs. This explains the effective heuristics developed by many earlier studies. On the other hand, the optimal placement is still unreachable for arbitrary access patterns. The paper shows a division between a few solvable or approximable sub-cases and the general case governed by the Petrank-Rawitz theorems.

2. Reference Affinity

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 . We use sequence and trace interchangeably.

DEFINITION 1. Volume distance. *The volume distance between two accesses, a_x and a_y , at times $T[a_x]$ and $T[a_y]$ in a trace T , is one less than the number of distinct data elements accessed*

in times between (and including) $T[a_x]$ and $T[a_y]$. We write it as $dis(a_x, a_y)$.

According to the definition, $dis(a_x, a_x) = 0$ and $dis(a_x, a_y) = dis(a_y, a_x)$. In addition, the triangle inequality holds— $dis(a_x, a_y) + dis(a_y, a_z) \geq dis(a_x, a_z)$, because the cardinality of the union of two sets is no greater than the sum of the cardinality of each set. For example, in the trace *abbbc*, the volume distance from the first a and to the last c is 2 and vice versa. The symmetry is important because the closeness is the same no matter which access happens first. Next we define the condition that a group of data elements are accessed close together.

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

In words, a linked path has a sequence of hops, each hop lands on a different data element and has a volume distance no greater than k . We call k the *link length*. We will later restrict the hops, x_1, x_2, \dots, x_t , to be members of some set S and say that there is a k -linked path from a_x to a_y with respect to set S .

For example consider the first part of the trace in Figure 1(a), *wxwxuyz*. The closeness between the first w and the last z is defined by the linked path with a minimal k , which is the path that jumps to the second x and then steps through each one in *uyz*. Each hop has a volume distance of 1 so is the link length. If we restrict the path to the set $\{w, x, y, z\}$, the link length becomes 2 since any path has to jump over u .

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

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

Reference affinity is a communal bond. All members of an affinity group must be accessed in the trace wherever one member is accessed. Each access is linked to some access of every other member in the group, and the linked path can go through only members of the group. We can now explain the hierarchy in Figure 1 fully. When $k = \infty$, any access in the trace is linked to any other access, so all data belong to one group. When $k = 0$, no two accesses can be linked, so each data element is a group. Now consider the group $\{w, x, y, z\}$, which are access in both parts of the trace. Its link length is 3 because in trace *zyyzvwxw*, no linked path can wade from the first z to an access of x without hopping through four different data elements around v . The path cannot land on the second z because it starts from z . Neither can it land on v because it is not a member of the group. When we reduce k , the group $\{w, x, y, z\}$ is partitioned into two sub-groups with closer affinity.

The initial purpose of this complex definition is for reference affinity to give a unique and hierarchical partition of data, as shown in the following three properties [44].

1. **Unique partition** Given an address trace and a fixed link length k , the affinity groups form a unique partition of program data.

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

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

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

DEFINITION 5. Finding reference affinity groups Given an address trace and k , find all reference affinity groups with link length k .

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

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

3. Hardness of Finding Reference Affinity

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

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

We prove it by making a polynomial-time many-one reduction from a variant of 3-SAT problem, where every variable appears at most three times (an NP-complete problem, see, e.g., [31]) to the linking problem. The proof constructs a three-part reference trace. The first part forces a linked path to go through a set of elements we call “separators”, which cannot be used as hops in the next two parts. The second part prepares a set of data triples to model the truth values of the logical variables in a 3-SAT expression. Since two data elements may represent opposite values of a logical variable, the construction ensures that the elements cannot be both included in a possible linked path. A linked path can land on different places and can even go backwards—the only constraint is that the volume distance of the longest hop. The critical moment of the construction is when the freedom of the linked path is contained within seven cases, and each is shown to have the needed property.

The third part of the sequence models all 3-SAT expressions. A linked path exists if and only if there is a truth value assignment to satisfy all expressions. To design a trace that enforces the logical consistency, we learned that we need to use multiple data accesses to represent logical variables instead of using data to represent them. The full proof is more than a page long and given in the appendix. From Theorem 1, we can easily prove two corollaries.

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

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

THEOREM 2. Pw-2-Aff is NL-complete.

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

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

ALGORITHM 1. Finding reference affinity groups when $k=2$

```

procedure FindReferenceAffinityGroup_2( $T$ )
1:  $\{T$  is the trace, the link length  $k = 2\}$ 
2: initially no affinity groups
3: while there exist elements not yet grouped do
4:   put all such elements into a set  $G$  and pick one  $x$  randomly
   from this set;
5:   repeat
6:     if there is an element  $z$  not 2-linked to  $x$  with respect to
        $G$  then
7:       remove  $z$  from  $G$ ;
8:     else
9:       if there exist two elements  $y, z \in G$  such that an
       access of  $y$  is not 2-linked to any access of  $z$  with
       respect to  $G$  then
10:        remove  $z$  from group  $G$ .
11:       end if
12:     end if
13:   until  $G$  is unchanged
14:   output reference affinity group  $G$ .
15: end while
endFindReferenceAffinityGroup_2

```

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

From Theorem 3, we can get the following corollary.

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

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

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

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

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

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

```

Compute( $D_1, D_2, \dots, D_n$ ) begin
  if the input data is above a threshold size
    divide  $D_1, D_2, \dots, D_n$  into sub-blocks
    for some set of sub-block combinations
      Compute( $subblock_i(D_1), subblock_j(D_2),$ 
        ...,  $subblock_k(D_n)$ )
    end for
    else process without sub-division
  end if
end

```

Figure 2. The general form of the divide-and-conquer algorithm

4. Reference Affinity in Divide-and-Conquer Computations

The divide-and-conquer type of computations we consider are blocked and recursive algorithms for dense matrix operations, N-body and mesh simulation, and wavelet transform. The general form is given in Figure 2. The procedure takes a set of data such as matrices. It then divides the input data into smaller blocks and processes all or subsets of their combinations. For each subset, if the blocks are still large, it makes a recursive call to itself. The computation is hierarchical, so is its locality.

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

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

In the following analysis, we assume a one-dimensional space. Higher dimensions can be linearized by using a space-fitting curve [30]. For simplicity, we assume that the space has $n = 2^t$ units, where integer t is non-negative. The N-body simulation trace is then of size 2^{2t+1} . As an example, we give the trace that follows the Morton space filling curve when computing the interactions between all pairs of four molecule data, shown in Figure 3 (a). The locality is in the recursive structure of computation. The data access trace is given in Figure 3 (b). We show that reference affinity can identify the locality structure by examining only the data access trace.

We call each data a unit and divide units into sections. We call the set of units $i * m + 1$ to $i * m + m$ an m -section of data, where m is a power of 2 and i is a non-negative integer. As the example in Figure 3 (a) shows, the rows and columns of the matrix are data units, and the Morton space filling curve gives the execution order. The interaction between an m -section and another m -section is computed at their product area in the graph, a block of size m by

m . We call it an m -block of computation. In a divide-and-conquer computation, each m -block contains a contiguous sequence of the computation trace.

We will prove the exact structure of the reference affinity hierarchy for divide-and-conquer computations. As a shorter exercise, we first show that the reference-affinity hierarchy has more than a constant number of levels when the size of data n is arbitrarily large.

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

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

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

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

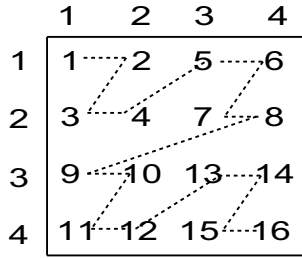
LEMMA 1. Insertion Lemma *Given two different data elements u and v ; their accesses a_u and a_v where the volume distance from a_u to a_v is exactly k ; and a third access a_x , which happens between a_u and a_v in the trace; then there exists an access a'_x between a_u and a_v such that the volume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

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

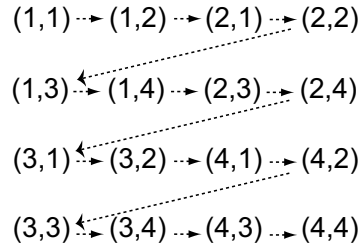
The following theorem gives the exact structure of the reference-affinity hierarchy. It is the most important theoretical result, establishing the link between the linear, flexible concept of linked paths in a computation trace and the hierarchical structure of locality in space.

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

The proof is straightforward after proving the following lemma.



(a) The Morton order for computing all interactions between all pairs of four molecule data



(b) The trace of the access to data

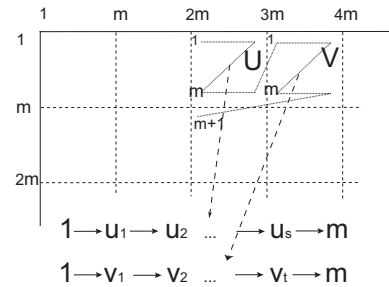
Figure 3. An example 4-body simulation

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

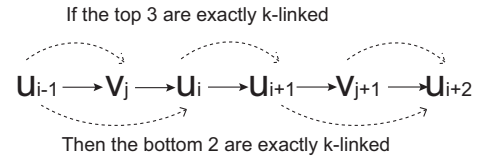
Proof Let k be the smallest reuse distance such that the m -section belongs to an affinity group. Without loss of generality, we assume the m -section and $2m$ -section are the first such sections in the data space, as shown in Figure 4(a). Suppose the $2m$ -section also belongs to the k -affinity group, we derive a contradiction by showing that the m -section belongs to a $k - 1$ -affinity group, for which it suffices to show that there is a $k - 1$ -linked path from the first access of 1 to the first access of m .

Because the $2m$ -section is in a k -affinity group, there is a k -linked path from the first access of element 1 to some access of element $m + 1$, as shown in Figure 4(a). The path is linked by at most one access of elements 2 to m . Now consider the two m -blocks of computation in the figure marked with U and V . They divide the path into two parts. By adding an ending point at the first access of m in the U block, and a starting point at the first access of 1 in the V block, we cut the k -linked path from 1 to $m + 1$ into two k -linked paths from an access of 1 to an access of m . The two paths are shown at the bottom of Figure 4(a). The intermediate links in the two paths are u_1, \dots, u_s and v_1, \dots, v_t . We map the v_i path in the V block to the U block. We now have two k -linked paths from the first access of 1 to the first accesses of m . The links are accesses to different data elements.

We construct a $k - 1$ linked path from the first access of 1 to the first access of m in U block in Figure 4(a). Consider each link on the u_i path, say from u_i to u_{i+1} . If the link length is not exactly k , then we are done. If the length is k , and some v_j happens in between, then from the insertion lemma, the k -link can be divided into two shorter links by moving v_j . If no v_j happens between u_i and u_{i+1} , there must exist v_j and v_{j+1} that include u_i and u_{i+1} in between. Since the volume distance from u_i to u_{i+1} is k , v_j and v_{j+1} must appear between u_{i-1} and u_{i+2} , forming the sequence shown in Figure 4(c). If the volume distance from u_{i-1} to v_j is smaller than k , then using the insertion lemma, the link from v_j to u_{i+1} can be divided into two smaller links by moving u_i , and the link from u_{i+1} to u_{i+2} can be divided into two smaller links by moving v_{j+1} . Similarly we can construct smaller links when the volume distance between v_{j+1} and u_{i+2} is less than k . Otherwise, $u_{i-1}, u_i, u_{i+1}, u_{i+2}$, are exactly k -linked. We continue to consider elements of $v_{j-1}, v_{j-2}, \dots, v_1$ and $v_{j+2}, v_{j+3}, \dots, v_t$ through similar steps. If we can not get a $k - 1$ linked path after examining all elements, it means that the path $1, u_1, \dots, u_s, m$ are exactly k -linked. This is impossible, since the original linked path



(a) Given that a k -link path exists from 1 to $m + 1$, we want to show that a $(k - 1)$ -link path exists from 1 to m . The k -link path is broken into two and placed in parallel at the bottom.



(b) When no links of length k can be broken into smaller links, all links must have exactly the length k , and the last link must be $k + 1$, which contradicts to the assumption.

Figure 4. An illustration for the proof of Separation Lemma

goes from the first access of 1 to an access of $m + 1$. The last link connecting to the access of $m + 1$ must have a length greater than k . A contradiction. ■

We make two observations. First, the proofs do not assume what, when, and whether a section of data is used. It requires only that a section is used together as a block. In N -body simulation, the interactions are calculated within a radius. In this case, an affinity

group cannot cross the boundary of a radius. The proofs assume the Morton order for the convenience of illustration, but it remains valid as long as all data are traversed in some order. The order may change when the same block is accessed at a different time. Hence the theorems can be extended to general divide-and-conquer algorithms. Second, the proofs are for the existence of k . The exact size of the data sections may change the value of k but not the existence of k . Therefore, the affinity hierarchy exists when data sections are divided into non-uniform sections.

Given Theorem 1, a natural question is whether the reference affinity in divide-and-conquer algorithms can be efficiently discovered. While the answer requires a systematic study that is beyond the scope of this paper, we note that our initial experiments in Section 6 show good results for recursive matrix multiplication. One reason is that in divide-and-conquer algorithms, the elements of the same affinity group are accessed in a similar order, while the reduction in the NP-complete proof requires data be accessed in all possible orders.

5. Affinity Analysis Through Sampling

Given a trace, an affinity group G , and $x, y \in G$, if there is a window of the trace covering at least an access for a_x and an access for a_y , and the length of the section is no greater than $k(|G|-1)+1$, then we call it a *witness window* for x and y . We consider that affinity holds for a_x and a_y if there is a witness window¹.

A sampling window of size w is a window of the trace where the volume distance between the two boundaries is w . We estimate affinity groups by finding elements that are frequently accessed together in sampling windows.

The sampling method is given by Algorithm 5. It first estimates the upper bound for the size of affinity groups. Suppose it is g . The size of the sampling window is set accordingly to $l = 2kg$. For a pair of data elements, x, y , the sampling method measures the affinity by what percent of the sampling windows have both x and y . Since they may appear in a different number of windows, the smaller number is used as the denominator. If the value is bigger than some threshold θ' , then we consider x and y have affinity. The pairwise relation is extended into a group relation by taking the transitive closure. To reduce the number of data elements considered in the analysis, we may exclude infrequently accessed data element, i.e. the number of appearances fewer than ϵ , in a similar fashion as association rule mining [21].

ALGORITHM 2. Sampling method for reference affinity analysis

Input: A trace; window size w ; sample rate δ ; threshold ϵ ; Affinity threshold θ .

Output: the reference affinity groups.

Method:

Let S be the number of sampled windows every single data element appears;

Let P be the number of sampled windows where each pair of data elements appear;

Sample windows of size w from the given trace, according to the sampling rate δ .

for each window do

for each distinct data element x do

increase $S[x]$ by 1.

end for

for every distinct pair of data element x, y do

Increase $P[x, y]$ by 1

end for

end for

¹This is an approximation because the bounded appearance is a necessary but not sufficient condition for reference affinity.

Ignore those data elements x with $S[x] < \epsilon$.

Construct a graph with the data elements not ignored as vertices.

for two vertices x, y do

if ($\text{confidence}(x, y) = P[x, y] / \min(S[x], S[y])$) $> \theta$

then

Add an edge between x and y

end if

end for

Output every connected subgraph as a group.

Suppose M is the number of distinct data elements, L be the length of the trace. The time complexity of this algorithm is $O(L\delta w^2)$. The space complexity is $O(M^2)$.

THEOREM 8. For any data element x in the reference affinity group, there exists a y in the same group and their expected confidence (defined in the algorithm 2) is greater than $\frac{1}{2}$.

Proof Suppose the upper bound for the affinity group size is g . Consider reference affinity group G . Clearly, $|G| \leq g$. For any data access to $x \in G$, from the definition of reference affinity, we can find an access to $y \in G$, where their volume distance is within $k(|G|-1)+1$. The sample size is $2kg$. Hence the sampling window has

$$1 - \frac{k(|G|-1)+1-1}{2kg} \geq 1 - \frac{k|G|}{2kg} \geq 1 - \frac{kg}{2kg} = \frac{1}{2}$$

probability of covering x and y , given it covers x .

Since the windows are sampled independent of x and y , the confidence value for x and y is at least $\frac{1}{2}$. ■

By Theorem 8, we know that if we set the threshold θ to be $\frac{1}{2}$, we can ensure the data elements in the same group remain in the same group found by Algorithm 2. Notice that in the algorithm, the window size, instead of the confidence threshold, is dependent of the reuse distance k .

The previous algorithm finds one level of reference affinity. For practice use, we can vary the sampling window size w in a logarithmic scale to find affinity groups at different levels.

Pairwise affinity has been used to reduce cache conflicts for code and data. Thabit [38] modeled the pairwise affinity of arrays in a proximity graph and showed that optimal packing is NP-hard. Gloy and Smith [19] studied procedure placement and used profiling to find the frequency a pair of procedures are called within a distance smaller than the cache size². Similar profiling methods are used for placing dynamic program data by Calder et al. [5] and Chilimibi et al [9].

The goal of the sampling method is to find reference affinity rather than to minimize cache conflicts. It is unique in at least three aspects. First, the pairwise frequency is the percentage of accesses. Consider the access sequence $abcabc..abc$. The percentage weight between all three data is 100%, so they belong to an affinity group. Now consider the sequence $abab..ab \dots bcbc..bc \dots acac..ac$. The percentage frequency is no more than 0.5 on average, so they do not have affinity, despite that access frequency of data pairs can be arbitrarily high. Petrank and Rawitz showed that for any placement method, a trace exists that has the same pairwise frequency but the data layout given by the method is at least a factor of $k-3$ away from the optimal layout, where k is the cache size. The sampling method alleviates this problem to a degree because the high percentage from the first example implies the affinity pattern in the access sequence, while the frequency from the second example does not contain enough information to ensure a good data layout. This shows the gist of the theory of reference affinity—it identifies a

²If a procedure p is called and the reuse distance after the previous call p' is no greater than twice of the cache size, the frequency is incremented for all pairs between p and all procedures called between p and p' .

specific access pattern and provides a better solution to the limited problem.

Two other differences are also significant. The size of the sampling window is proportional to the group size rather than the cache size. Since the useful group size is the size of cache blocks, it is much smaller than the cache size, and the smaller window allows more efficient analysis and in turn larger traces and finer granularity. Finally, the data transformation is simple, which is to group members of the same affinity group. It is no problem if group members have different sizes. In contrast, previous methods solve weighted data packing problems and must reconcile between different size data. The difference is due to the fact that reference affinity is transitive but the pairwise frequency is not. The placement between affinity groups is still a problem of packing, and the general complexity is given by Petrank and Rawitz. Still, the solution within an affinity group is clear.

As a heuristic, the sampling method is not accurate for several reasons. First, it may miss an affinity group when not enough witness widows are sampled. Second, it may find false groups that are not accessed together as often as they do in sampling windows. While in general one cannot guarantee without a priori knowledge about the distribution of data accesses in a trace, we can use a higher sampling rate to improve the statistical coverage and accuracy. In fact, we can sample every window in profiling analysis. Finally, it needs an upper bound for the group size. This prevents the method from finding large affinity groups. However, if we target specific hardware, the size of the storage unit, for example, the size of cache blocks, can be used as the upper bound for the group size.

6. Evaluations of The Sampling Method

6.1 Comparisons with K-distance Analysis

We first review the k-distance analysis based on reuse signature [44]. K-distance analysis targets only groups of data that are always accessed together. It first measures the reuse signature of every data element, which is the histogram of the reuse distance of all accesses of the element. Then it computes the Manhattan distance between the reuse signature of every data pair x, y as follows.

$$d = \sum_{i=1}^B |Avg_i^x - Avg_i^y| \quad (1)$$

where Avg_i^x is the average distance for bin i , B is the number of bins considered. If their access pattern is near identical, $|Avg_i^x - Avg_i^y|$ is smaller than k in every bin. Hence if $d \leq k * B$, then x, y are in the same affinity group.

K-distance builds the affinity hierarchy incrementally. Initially every data element is a group. Then it merges two groups if the distance between a member in one group and a member in the other group is the smallest among all cross-group distances.

Compared with the sampling method, k-distance tends to cluster irrelevant data elements into one group, especially for those single data elements which occur randomly. The second problem is that the vector may not be the same for data elements in the same strict reference affinity group because of the partition boundaries when collecting the histogram. Finally, it does not work well for partial reference affinity where groups of data are often but not always accessed together. Their reuse signatures may differ by an average of more than k .

We first compare the two methods using generated traces. A direct measure is the number of perfect matches between the affinity groups we use to generate the trace and the affinity groups found by an analysis method. We call it the *match rate* of the analysis. A more complex metric, *accuracy*, measures the quality of imperfect matches. Let an affinity group G be separated into pieces

P'_1, P'_2, \dots, P'_n and scattered into groups G'_1, G'_2, \dots, G'_n found by an analysis method. We define the accuracy for this affinity group as

$$accuracy(G) = \frac{\sum_{i=1}^n \frac{|P'_i|}{|G'_i|}}{n^2}.$$

The more pieces a group is separated into, the lower the accuracy is. Same is true when a small group is clustered into a bigger group. We use n^2 instead of n as the denominator. Consider the following situation: if G is scattered into exactly $|G|$ trivial groups, then the accuracy would be 1 if we used n as the denominator. The overall accuracy is the average accuracy of all affinity groups. We use both the match rate and the accuracy in the following comparison. When the affinity groups are correctly recognized, both the match rate and the accuracy are 1. Otherwise the accuracy is in general higher than the match rate because the former includes partially recognized groups.

The results presented here are an average of simulating 20 traces. The variances of the accuracy and the miss rate are very small and we don't report them here. The size of the trace is 200,000 by default or a length in which every affinity group occurs roughly 400 times. In every figure except Figure 7, the link length k is set to 1, which is the best case for k-distance analysis. Figure 7 will show that when k increases, the accuracy of k-distance drops much faster than the sampling method does. For every experiment, the sample rate is set to be 0.001. For lack of space, we omit parameters of the trace generation that are not essential to the presentation. A more detailed description can be found in a technical report [42].

We first compare sampling and k-distance analysis on 50 groups whose members are often but not always accessed together. The frequency of the affinity is the weakness. A weakness of 0.7 means that when one member is accessed, 70% of other members will be accessed. Figure 5 shows that when the weakness is 0.5, the match rate and the accuracy are over 96% for the sampling method but below 73% for k-distance analysis (the accuracy is under 65%). As the weakness changes from 0.5 to 1, both methods improve. The match rate and accuracy are close to perfect for the sampling method but still below 86% for k-distance analysis.

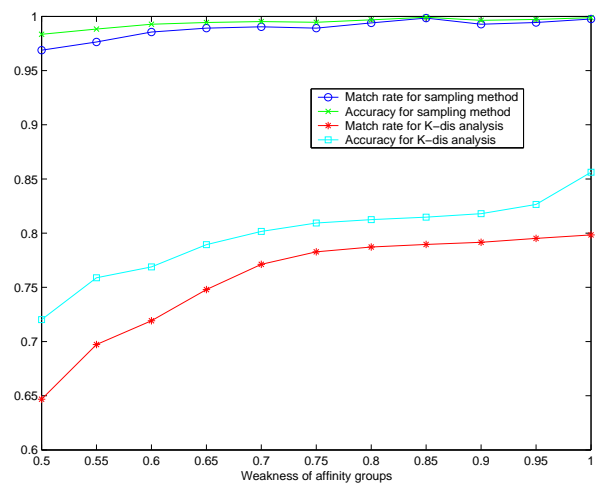


Figure 5. Comparison of performance when groups of data are often but not always accessed together

Figure 6 shows the comparison when the number of affinity groups varies. For strict affinity groups (weakness is 1), when the number of groups change from five to fifty, the match rate and the

accuracy are stable (around 85%) for the sampling method but drop significantly for k-distance, from 86% to 67% for the match rate and from 78% to 50% for the accuracy. The match rate of the sampling method is not only much higher but also much closer to the accuracy, compared with k-distance. The upper two curves differ by no more than 5%, showing that most affinity groups are detected in whole by the sampling method.

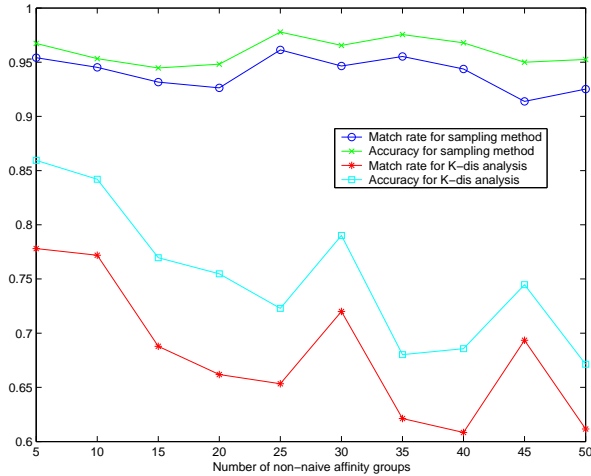


Figure 6. Comparison of performance when the number of affinity groups varies

We now compare the effect of k , which is the closeness between accesses to data of the same affinity group. As k increases, the complexity of finding affinity groups increases, from polynomial time to NP-hard as shown in theory in earlier sections. Figure 7 shows the result for 200 strict affinity groups. The match rate of the sampling method is perfect when k is 1 and 2 and drops to around 80% when k increases from 3 to 10. K-distance analysis has much worse performance from the beginning (around 85%) followed by a steep drop (to below 20%). The sampling method detects most affinity groups in whole but k-distance does not.

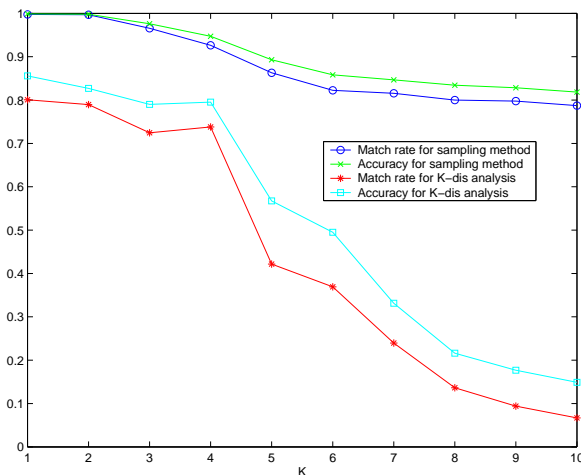


Figure 7. Comparison of performance when k (the closeness) of the affinity group varies

Last we test different sizes of the sampling window using affinity groups of size 20. Figure 8 shows that the performance is best when the window size is 20, comparable to kg , where $k = 1$ and

$g = 20$. Since accesses are randomly scattered, the average distance is about half of kg . Thus using only kg , we can get enough confidence. The performance varies by less than 2% for window sizes between 10 and 30 and less than 8% between 5 and 50. The match rate and the accuracy are always higher than 92%. Therefore, the sampling method tolerates a wide range of window sizes when targeting groups of specific sizes.

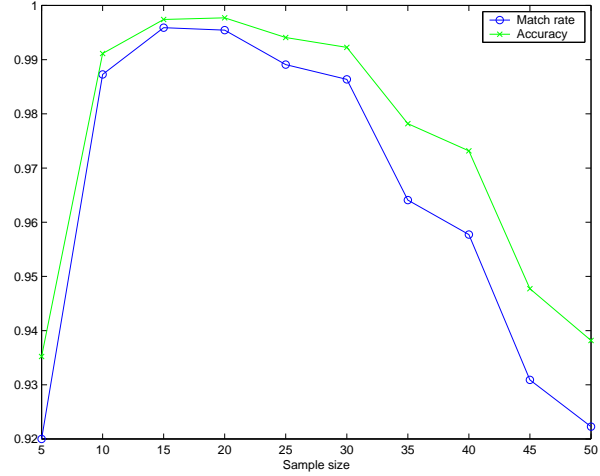


Figure 8. The performance of the sampling method for different window sizes

The two methods can be combined. We first run K-distance analysis to find approximate affinity groups, estimate the size of the affinity groups, and then use the sampling method to refine and improve the results. The estimate from K-distance analysis also reduces the memory requirement of the analysis. We have left out results from other experiments based on generated traces because of the limited space. In all cases, the sampling method gives efficient and accurate analysis for a wide range of affinity groups. Next we look at real program traces.

6.2 Recursive Matrix Multiplication

We test affinity analysis on recursive matrix multiplication. Given two square matrices, each has 256 elements (16×16), the program recursively divides the matrices into four parts and calculates their product. The length of data access trace is 12288 (3×16^3). According to Theorem 7, there are five levels of reference affinity in 16×16 , 8×8 , 4×4 , 2×2 , and 1×1 sub-matrices. When we set the sampling window to be volume distance 12 and the affinity threshold to be 0.6, the sampling method correctly identifies affinity groups in 2×2 blocks. Figure 9 shows one of the input matrix, with affinity groups drawn in different shades of gray. Figure 10 shows that k-distance analysis identifies half of the 2×2 blocks but groups others into 4×4 blocks, which is not as accurate as the sampling method.

This is a simple experiment, but it demonstrates that sampling analysis can uncover the high-level locality structure despite that it examines only the data access trace of the complex computation, and that the general problem is NP-hard.

The affinity analysis can be used in a profiling tool for a user to see the locality structure in program data. Some of the data transformations can be automated, for example, structure splitting and array regrouping. Some cannot, for example, the recursive data layout in complex computations, so user support is currently needed. It may be possible for dynamic data transformation. A program can often analyzes its run-time data access and reorganizes the data for better locality, as demonstrated by many studies for array data in scientific programs or objects managed by a garbage

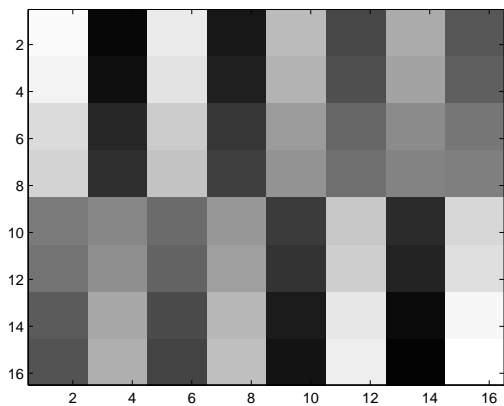


Figure 9. All 2*2 affinity groups are identified by the sampling method

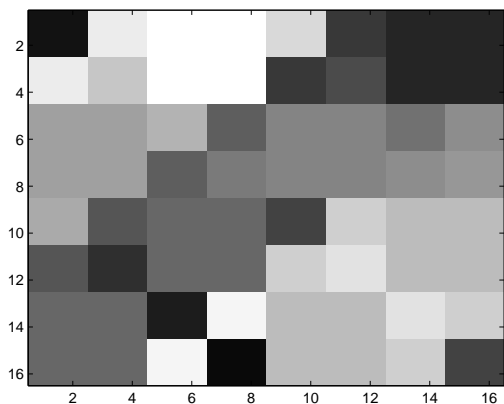


Figure 10. Half of the 2*2 affinity groups are identified by k-distance analysis

collector. Reference affinity should be profitable if its analysis can be made efficient enough.

6.3 Affinity-based Code Layout

We use profiling to collect the trace of basic block references, apply reference affinity analysis, and then reorganize code layout by placing basic blocks of the same group sequentially in memory. The *sim-cache* module of SimpleScalar is modified to simulate the instruction cache and measure the cache miss rate. We compare these affinity groups with the original code layout and the code regions formed by checking transition frequencies between basic blocks [22], a common method in profiling-based code layout. For lack of space, we briefly describe only the main setup and the result. The purpose is to demonstrate the applicability of reference affinity and the sampling method. We do not intend in this work to design a complete compiler technique nor evaluate it against the existing literature.

We test seven integer programs from SPEC 2000³. We use complete traces, which contain up to one billion basic blocks and 10 billion references. K-distance method is based on the reuse signature

³ *Bzip2, Crafty, Gap, Mcf, Perlbnk, Twolf*, and *Vpr*

from long-distance reuses, so it ignores many basic blocks, which are only accessed once with a short reuse distance. It suffers more from false positives because the reuse signature lacks the timing information. In experiments, k-distance tends to produce one or two very large groups, containing up to 50% of all basic blocks. The sampling method overcomes these drawbacks. For complete coverage in profiling, the sampling rate is 100%. The exact comparison depends on the thresholds and parameters for both sampling and the frequency-based method. A detailed comparison is too long to include. However, the best affinity-based layout always has better locality than the best frequency-based layout, as shown in Figure 11 for two different cache configurations. Compared with the unoptimized code layout on 8KB direct-mapped cache, the affinity-based layout reduces the cache miss rate for six of the seven programs by up to a factor of more than 3. When 16KB 2-way set associative cache is used, the miss rate of the three of the seven programs drops to near zero. The affinity-based layout improves the locality in all other four programs. The largest relative improvement is on *Twolf*, a circuit placement and global routing program using simulated annealing. The miss rate drops from 0.4% by more than a factor of 10 by the affinity-based layout. In comparison, the frequency-based layout reduces the miss rate by a factor of about 2.

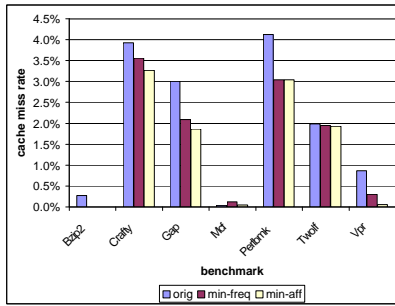
7. Related Work

Thabit showed that data packing for a given block size using pairwise frequency is NP-hard [38]. Kennedy and Kremer gave a general model that includes run-time data transformation (among other techniques) and showed that the problem is NP-hard [23]. Petrunk and Rawitz showed the strongest theoretical result to date—if $P \neq NP$, no polynomial time method can guarantee a data layout whose number of cache misses is within $O(n^{1-\epsilon})$ of that of the optimal data layout, where n is the length of the trace. In addition, if only pair-wise information is used, no algorithm can guarantee a data layout whose number of cache misses is within $O(k - 3)$ of that of the optimal data layout, where k is the size of cache. The results hold even when the computation sequence is completely known, objects have the same size, and the cache is set associative [32]. These general results, however, do not preclude effective optimization targeting specific (rather than all) data access patterns. In fact, one can easily construct traces for which it is trivial to find the optimal data layout.

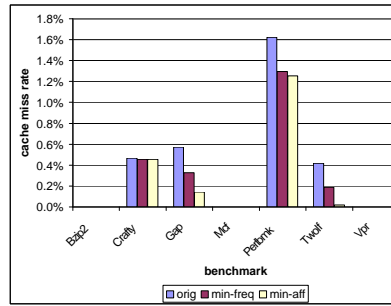
Zhong et al. defined reference affinity and used a heuristic called k-distance analysis in structure splitting and array regrouping [44]. The earlier work does not give the computational complexity of reference affinity, nor is it used in hierarchical data placement. In addition, as shown in Section 6, k-distance analysis does not work well for partial reference affinity.

Hierarchical data layout is first used for matrix multiplication and QR factorization by Frens and Wise [16, 17], Cholesky factorization (based on data shackling [25]) and wavelet transform by Chatterjee et al. [7], and N-body and mesh simulation by Mellor-Crummey et al [30]. We show that reference affinity can help a programmer to analyze data locality in these and other programs (without knowing the structure of the computation).

Bender et al. used the recursive van Emde Boas layout for dynamic search trees and proved the asymptotic optimality if the input consists of random searches [3]. Reference affinity cannot yet give the van Emde Boas layout because the affinity hierarchy is flat (two levels) for any constant k . The extension for variable-distance affinity groups is a subject of future work. On the other hand, if the input is not random search and has reference affinity, for example, a group of tree nodes are often searched together, then reference affinity is directly applicable and in principle may yield better locality than the van Emde Boas layout does.



(a) 8KB direct-mapped cache, cache block size 64 bytes



(b) 16KB 2-way set associative cache, cache block size 32 bytes

Figure 11. Comparison of cache performance of the original and the best of frequency- and affinity-based code layout for seven SPEC2K integer benchmarks

Wolf and Lam [40] and McKinley et al. [29] used compiler analysis to identify groups of memory references in loop nests. Our work is related to trace-based pairwise affinity models, as discussed in Section 5. Hang and Tseng used pairwise (connection) models for scientific data and found that hierarchical clustering was more cost-effective than general graph partitioning [20]. Chilimbi defined *hot data streams*, which are sequences of repeated data accesses and are not limited by pairwise affinity [8]. When a program has good computation locality, simple data packing (first-touch ordering) improves the data layout, as shown by several studies [12, 30, 36, 39]. Recently, Strout and Hovland introduced models based on hyper-graphs [37]. Most of these techniques are intended for on-line adaptation and do not model hierarchical locality.

Earlier studies have established the locality upper bound (the best possible locality) for specific computing problems. Hong and Kung used a graph model to prove the tight lower bound of I/O for FFT, matrix multiply, and sorting. Aggarwal et al. gave a hierarchical memory model (HMM), where the cost for access location x was $\lceil \log x \rceil$ [1]. They showed that the maximal slowdown factor ($O(\log n)$) could be avoided for some computations (FFT, matrix multiply, and sorting) but not for others (list search and circuit simulation). They also studied other convex cost functions. Alpern et al. extended the hierarchical model with explicit transfer cost and gave a set of threshold functions when computations (FFT, matrix multiply, matrix transpose, and parallel matrix multiply) became communication bound [2]. Frigo et al. refined the cache model to consider cache size Z and block size L . They gave recursive algorithms that yielded the best asymptotic locality for FFT, matrix multiply, and sorting for any Z and L ($Z = \Omega(L^2)$) [18]. The algorithms are cache oblivious because they do not depend on Z and L . Yi et al. showed that a compiler can automatically convert loops into recursive subroutines to exploit hierarchical locality [41].

The ordering of general computations has been studied as a clustering problem in loop fusion and computation regrouping. Kennedy and McKinley [24] showed that fusion for locality was NP-hard because it could be reduced to the problem of k -way cut [10]. Ding and Kennedy proved a similar result for a refined model, where hyper graphs were used to represent data reuses among multiple computations [13]. Darté gave complexity results for a larger class of fusion problems [11]. These results suggest that efficient, hierarchical models are unlikely for general-purpose computations.

In 1970, Mattson et al. first defined reuse distance (named LRU-stack distance) [28]. The distribution of all reuse distances (which we call the reuse signature) gives the temporal locality of an execution. Snir and Yu recently disproved the possibility of a more compact metric by showing that temporal locality could not be characterized by one or few parameters [35]. Indeed, reuse signature has been widely used in virtual memory and cache design. Building on decades of development by others, our earlier work reduced the measurement cost to near linear time [14] and used reuse distance patterns for program locality prediction [33, 43], data structure transformation [44], and locality phase prediction [34]. Reuse-distance based models are found useful for program analysis by an increasing number of studies (three appeared in 2005), including cache miss rate prediction across inputs by Marin and Mellor-Crummey [26, 27], per-reference miss rate prediction by Fang et al. [15] and Beyls and D’Hollander [4], and cache interference prediction for parallel processes by Chandra et al [6]. This paper presents a formal theory for reuse distance. The proof for Theorem 1 gives a reduction between reuse distance and formal logic. The proof of Theorem 7 connects the structure in computation with the locality in data. These formal connections serve as a theoretical basis for current and future reuse-distance based models and techniques.

8. Summary

In this paper we have given a complete characterization of the complexity of reference affinity. We have proved that when $k = 1$ or $k = 2$, finding reference affinity groups can be done within polynomial time and when $k = 3$, finding reference affinity groups is a NP-hard problem. We have extended the hierarchical locality to divide-and-conquer computations such as matrix solvers, factorization, wavelet transform, N-body and mesh simulation, where the results confirms the previous empirical solutions. The theoretical results have established formal links between the computation, the data reuse, and the locality. In practical use side, We don’t know a real use for cases $k = 1$ or $k = 2$. The hardness of finding the reference affinity groups when $k > 2$ implies that it is hard to find any polynomial algorithms. Instead, we have presented a sampling method and shown through experiments that it is more accurate than the previously published technique, especially for groups greater in number and complexity and weaker in their affinity. We have shown two new uses of reference affinity. The first is finding

hierarchical data layout in a recursive program, and the second is improving the code layout of seven SPEC 2K applications.

In POPL 2002, Petrank and Rawitz precisely characterized the theoretical difficulty of the general data placement. Reference affinity side steps this limitation by targeting a common pattern rather than all patterns of data access. Since the volume distance is widely used in experimental algorithms, the theoretical findings in this paper may help the development of other distance-based locality theories.

A. Proofs

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

Proof It is obvious that the problem is in NP. We will prove its NP-hardness by constructing a polynomial-time many-one reduction to Pw- k -Aff from 3-SAT, which is the problem of testing, given a formula of conjunctive normal form in which each clause has at most 3 literals, whether the formula is satisfiable. We consider the variant of this problem in which each variable appears as a literal (positively or negatively) at most three times. This problem is also known to be NP-complete (see, e.g., [31]). Without loss of generality, we can assume that all variables appear both positively and negatively in the formula. If a variable appears only positively (respectively, negatively) then we can create a simpler, equivalent formula by setting the value of the variable to *true* (respectively, *false*).

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

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

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

The sequence Σ is the k repetitions of $\nu_1 \cdots \nu_k$ separated by $k-1$ λ_{in} 's, where ν_1, \dots, ν_k are k pairwise distinct labels. Recall that for a pair of positions to be k -linked there must be a set of intermediate points with pairwise distinct labels in which the reuse distance between each neighboring intermediate points is at most k . To create such a path between our two end points, the subsequence Σ must be traversed without visiting a same label more than once so that the distance between the two neighboring visited points have reuse distance at most k . The only way to construct such a path is to visit every $(k+1)^{st}$ element of Σ besides the first λ_{in} , exiting at the first element after Σ . This path visits ν_1, \dots, ν_k exactly once. This means that any k -link path between our two endpoints should not visit any one of ν_1, \dots, ν_{k+1} again.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

On the other hand, suppose that φ is not satisfiable. Take any potentially k -linked path π in the Θ 's. There exist at least one variable, x_i for which both one occurrence of x_i and one occurrence of $\overline{x_i}$ is selected. Then it is not possible to construct a k -linked path within Γ_i , so there is no k -linked path between the two endpoints.

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

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

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

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

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

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

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

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

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

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

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

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

Suppose that this formula is satisfiable. Let A be a satisfying assignment of the formula. Then A clearly defines a k -linked path, since only those belonging to Σ are visited, no element in Σ is visited more than once, and there is at most one entry between any

two neighbors on the path. Similarly, if there is a k -linked path, then by setting the truth value of each variable according to whether the node is included in the path, we can satisfy the formula. So, the satisfiability of the formula is equivalent to the existence of a k -linked path. ■

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

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

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

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

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

$$\dots y \dots l \dots x \dots z \dots \quad (2)$$

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

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

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

ALGORITHM 3. Finding reference affinity groups when $k=1$

procedure FindReferenceAffinityGroup_1(T)

1: $\{T$ is the trace, $k = 1\}$

```

2: encode the data elements according to the order of appearance
   in the trace. Suppose there are  $m$  distinct data elements.
3: while there exist data not yet grouped do
4:   pick the smallest not yet grouped datum  $s$ .
5:   for  $t=m$  to  $s$  step  $-1$  do
6:     if  $IsAGroup(T,s,t)$  then
7:       break;
8:     end if
9:   end for
10:  output elements in  $\{s, \dots, t\}$  as a group.
11: end while
endFindReferenceAffinityGroup_1
procedure  $IsAGroup(T,s,t)$ 
1: for  $i$  from 1 to  $|T|$  do
2:   if  $T[i]$  is within  $s$  and  $t$  then
3:     if The elements  $T[i]$  can be 1-linked to with respect to
        $\{s, \dots, t\}$  can not cover set  $\{s, \dots, t\}$  then
4:       return false;
5:     end if
6:   end if
7: end for
8: return true;
endIsAGroup

```

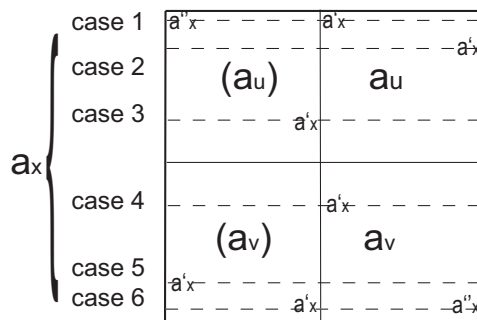
It is straightforward to show that the algorithm is polynomial time and can output the correct reference affinity groups.

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

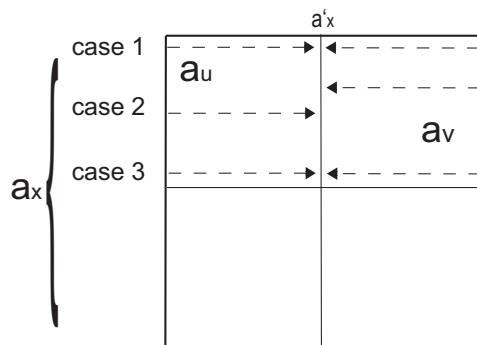
Proof The element u is either earlier or later than v in the data space. Because a link and a path are not directed, the two cases are symmetrical. Without loss of generality, we assume u is before v . Consider the smallest m -block that contains u, v, a_u, a_v . The element x must be in the data section of the block; otherwise the path from a_u to a_v does not go through x . There are two cases shown by the two graphs in Figure 12, each has six sub-cases. The location of a_x is given for each sub-case in the figure. In most cases, a_x splits the k -link from a_u to a_v into two shorter links of less than k . The first case is when a_u and a_v are in upper and lower half blocks. In the first sub-case of the first case, we need to use one of the two locations, marked by a'_x and a''_x . Then we use the a_u to break the link from the access of x to a_v and treat the access to x as a_u . The last sub-case of the first case is similar. The second case happens when a_u and a_v are both in the upper or lower half block. Figure 12 shows the three out of the six sub-cases when both accesses are in the upper half block. The other three sub-cases are symmetrical. In sub-case 1 and 3, we pick a'_x to be in the middle on the same side of a_x . In sub-case 2, we use one of the two middle points depending on the position of a_x . ■

References

- [1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the ACM Conference on Theory of Computing*, New York, NY, 1987.
- [2] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, 1994.
- [3] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *Proceedings of Symposium on Foundations of Computer Science*, November 2000.



(a)



(b)

Figure 12. Cases in proving the insertion lemma. a_u and a_v show the possible positions in the trace. a'_x , and a''_x show the possible targets of moving x to split trace between a_u and a_v . The arrows show the destination of moving x . For every case, there is a solution for x to break the k -link in the old path.

- [4] 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.
- [5] 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.
- [6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2005.
- [7] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of International Conference on Supercomputing*, 1999.
- [8] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.
- [9] 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.

- [10] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, May 1992.
- [11] A. Darté. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.
- [12] 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.
- [13] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.
- [14] 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.
- [15] C. Fang, S. Carr, S. Onder, and Z. Wang. Instruction based memory distance analysis and its application to optimization. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, 2005.
- [16] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance with source code. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, Las Vegas, NV, 1997.
- [17] J. D. Frens and D. S. Wise. QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, San Diego, CA, 2003.
- [18] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of Symposium on Foundations of Computer Science*, October 1999.
- [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 scientific codes. Technical report, Department of Computer Science, University of Maryland, College Park, 2000.
- [21] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.
- [22] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. In *Proceedings of the Annual International Symposium on Microarchitecture*, 1995.
- [23] K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4), 1998.
- [24] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).
- [25] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.
- [26] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*, New York City, NY, June 2004.
- [27] G. Marin and J. Mellor-Crummey. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, 2005.
- [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] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.
- [32] 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.
- [33] 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.
- [34] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.
- [35] M. Snir and J. Yu. On the theory of spatial and temporal locality. Technical Report DCS-R-2005-2564, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, July 2005.
- [36] 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.
- [37] M. M. Strout and P. Hovland. Metrics and models for reordering transformations. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Memory System Performance*, Washington DC, June 2004.
- [38] K. O. Thabit. *Cache Management by the Compiler*. PhD thesis, Dept. of Computer Science, Rice University, 1981.
- [39] B. S. White, S. A. McKee, B. R. de Supinski, B. Miller, D. Quinlan, and M. Schulz. Improving the computational intensity of unstructured mesh applications. In *Proceedings of the 19th ACM International Conference on Supercomputing*, Cambridge, MA, June 2005.
- [40] 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.
- [41] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.
- [42] C. Zhang, Y. Zhong, C. Ding, and M. Ogihara. Finding reference affinity groups in trace using sampling method. Technical Report TR 842, Department of Computer Science, University of Rochester, July 2004. presented at the 3rd Workshop on Mining Temporal and Sequential Data, in conjunction with ACM SIGKDD 2004.
- [43] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.
- [44] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.