

A Method for Hierarchical Data Placement

Chengliang Zhang, Yutao Zhong, Chen Ding and Mitsu Ogihara

Computer Science Department
University of Rochester

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

Abstract

To fully utilize the hierarchical memory on modern machines, *hierarchical data placement* reorganizes program data in many layers of data blocks, which exploit data locality at *all* memory levels. Manually designed methods have been used in at least three important application domains—matrix operations, N-body simulation, and search trees—by different research groups using very different data layouts. This paper presents a new method for hierarchical data placement. The method is based on the reference affinity model, which captures the temporal pattern in program computation and converts it into the spatial relation in program data. The method is applicable to any sequential programs. It is polynomial time. It automatically gives the hierarchical data layout not only for cases reported by previous studies but also for other important problems. The new results reveal strong relations between computation and data and therefore should help to improve the management of data as well as the design of compilers and programming languages.

Keywords: hierarchical data placement, reference affinity, trace, N-body simulation, search tree

1 Introduction

While program data are laid out in a uniform address space, the memory of most machines is organized as a hierarchy, which typically includes registers, on-chip and off-chip cache, virtual memory, disk and network buffers. Different memory levels come with different sizes and configurations. The hierarchy may not be fully visible to a user. In addition, a program may be run on machines with different cache 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 [5], N-body simulation [?], and search trees [2], where the program data are recursively decomposed into smaller blocks. Such

hierarchical partition exploits even an unknown cache size because it blocks for all cache sizes.

Previous data placement methods are specialized: Morton layout for matrices, Hilbert curve for particles, and van Emde Boas layout for search trees. It is unclear whether these diverse methods share a common basis, whether more general methods exist, how about data placement in other problems, and ultimately, for a program, how to find the right data hierarchy.

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 computation traverses the data contiguously. We think of the data placement problem as the problem of mapping. The domain is the set of programs, which is the power set of the set of all sequences of data accesses. The image is all data decompositions. The mapping takes from a program to its hierarchical data placement, from a uniform address space to a data hierarchy, and from sequences in time to structures in space.

To place data is to define their spatial relation — some data but not others should be placed together. Three general models exist for defining the relation. The first is compiler analysis. By analyzing structured loops and subscripted array accesses, a compiler can often derive the best computation order and data layout. However, most compiler data models are linear rather than hierarchical. In addition, it is less effective for programs with input-dependent control flows and indirect data accesses. The second is profiling of the frequency of the access to data or data pairs. Although the frequency model measures the reuses of data, it does not consider the time of data reuse and consequently cannot guarantee a good data placement [18].

The last, and the most recent of the three is *reference affinity*, which we have been developing over the past five years. Reference affinity measures whether a group of data are always accessed together during an execution. We use the distance of data reuse, which is the amount of other data accessed between the use

and the reuse of a datum. The reuse distance is Euclidean. It combines the information of the frequency, the time, and the volume of data access. Our earlier papers described efficient analysis methods for reuse distance and reference affinity [9, 27] and showed their use in miss-rate prediction across program inputs, array re-grouping in Fortran programs, and structure splitting in C programs [26, 27]. The accuracy of the reuse-distance model has also been verified by other recent studies [3, 10, 15].

In this paper, we present a method for constructing a hierarchical data placement for a program. We call it the *data hierarchy method*. It divides the reference affinity groups into two cases: the groups with a constant distance and those with a variable distance. The construction of the data hierarchy involves a bottom-up traversal of the affinity hierarchy in the first case and a bottom-up partition of a probability graph in the second case. The next section describes the algorithm in detail and shows its use in hierarchical data placement in general and extreme cases. The section shows that the prior results in matrix multiplication, N-body simulation, and tree searches are just examples of the general case. As a unique use of the system, it gives the hierarchical data placement for random walks, whose data placement problem had not been studied in the literature.

We note here that the scope of this paper is limited to the data placement problem, that is, finding that the best data layout for a given computation sequence. Computation reordering is outside the scope. The construction method uses the result of reference affinity analysis. The analysis does not have a known solution in polynomial time, although good heuristics exist [25, 27].

As a result of hardware and software trends in the past decades, data organization has become increasingly important for program performance, communication speed, power consumption, and software portability. Programming for data is often as important and complex as programming for computation. A general model of hierarchical data organization is relevant to programming as well as to programming language design and implementation.

2 The Data Hierarchy Method

2.1 Basic Structure

We first define the necessary concepts and then give the basic structure of the data hierarchy method.

2.1.1 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 .

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

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

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

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

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

$$\dots abebc \dots bffaac \dots ccccbga \dots \quad (1)$$

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

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

Reference affinity groups give a unique and hierarchical partition of data, as proved by Zhong et al. in the form of the following three properties [27].

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

2.1.2 The Data Hierarchy Method

Given an execution trace and its data size n , we have a hierarchy of affinity groups parameterized on the link length k . When k is 0, we can have no linked path connecting elements; so each data element is an affinity group. When k is $n - 1$, we have a linked path from every element to every other element; so all the data belong to a single affinity group. For every k in between, we have some partition of the data. Therefore, we have an n -level affinity hierarchy, each level is a partition of the data. For any two consecutive levels, the partition at k is either the same partition as or a finer partition than the level at $k + 1$.

The data hierarchy method, shown in Algorithm 1, takes the affinity hierarchy and the data size as inputs. It first removes identical levels and splits the affinity hierarchy at the level whose link length k is greater than the largest cache size. The link length gives the longest volume distance between accesses to elements of the same group. If the group resides in a cache block and its k is smaller than the cache size, we guarantee the reuse of every group member after the block is accessed and before it is evicted due to a capacity miss. If k is greater than or equal to the cache size, however, we cannot guarantee the reuse of every group member. Because of this

difference, the method builds the hierarchy differently in two cases.

Algorithm 1 The method for constructing a data hierarchy

procedure *MainConstruction*(g, n)

{ g is the reference affinity hierarchy with levels 0 to $n - 1$, and n is the size of program data}

{Remove identical levels}

for k from 1 to $n - 1$ **do**

if the level at link length k is the same partition as the level at $k - 1$ **then**

 Remove this level from the hierarchy

end if

end for

Renumber the remaining levels from 1 to h

{Divide the affinity hierarchy into two parts}

Let i be s.t. the levels $1, \dots, i$ have a link length at most the largest cache size

Set $g_1 = 1, \dots, i - 1$ and $g_2 = i, \dots, h$

{process the two sub-cases and assemble the results}

$d_1 = \text{ConstantDistanceCase}(g_1, n)$

$d_2 = \text{VariableDistanceCase}(g_2, n)$

Stack the two data hierarchies d_1 and d_2 and return the result

end *MainConstruction*

The reference affinity is defined on an execution trace, so our method constructs the data hierarchy for a trace instead of a program, which is a set of execution traces. However, in some programs (as shown by examples later), the affinity hierarchy has a consistent structure parameterized by the input size n . We can discuss the affinity groups with a symbolic n . In this case, the division of the affinity hierarchy happens at the level where the link length k is a function of n instead of a constant independent of n . The reason is that when k is a function of n , it will become larger than any fixed cache size when n is sufficiently large. For this reason, we call the first half the *constant distance* case and the second part the *variable distance* case.

2.2 Constant-Distance Groups

We first consider the case in which the link length k of an affinity group is a constant. We put all members of the group in a data block. When k is less than the cache size, we guarantee that all elements are accessed after the block is loaded into cache and before it is evicted from cache due to a capacity miss. The accesses to the block elements keep the block “warm” in cache. Therefore, we guarantee the full block utilization for constant-distance affinity groups.

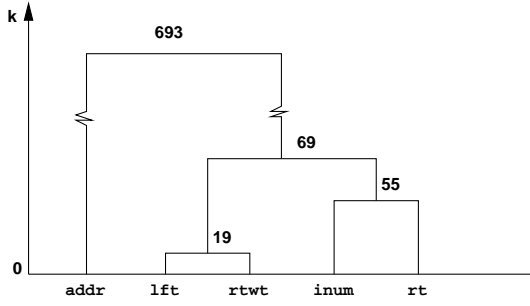


Figure 1: Dendrogram for Cheetah

Reference affinity can be shown as a tree or a dendrogram. Figure 1 [27] shows the reference affinity among five structure fields used in *Cheetah*. *Cheetah* is a hand optimized cache simulator for simulating multiple cache configurations in one run. It is widely distributed as part of the SimpleScalar tool-set. The main data structure is a splay tree. The five fields of a tree-node structure are shown in Figure 1: *addr*, *lft*, *rtwt*, *inum* and *rt*. The access to the tree is a sequence of queries, each consists of a top-down tree search and a bottom-up splay. The search involves the key (*inum*) and the children pointers (*lft* and *rt*). The splay involves the tree weight (*rtwt*) and the children pointers. The search path is stored in another data structure so a tree node does not need a parent pointer.

In this example simulation, the majority of the searches targeted the right-end of the tree. As a result, the left-child pointer and the node weight have the closest affinity because the bottom-up splay rotates to the left more often than to the right. The key and the right-child pointer have the next closest affinity because the top-down search follows the right-child pointer more often than the left-child pointer. The *addr* field is accessed only at the end of a query, so it has little affinity with the other four fields. The affinity hierarchy is shown by the dendrogram in Figure 1. The number marked at each non-leaf node is the link length of the group of elements in its sub-tree. For example, *lft* and *rtwt* are accessed mostly within 19 data elements.

For constant-distance groups, we construct a data hierarchy by traversing the dendrogram in postorder. Algorithm 2 gives an exact description.

Algorithm 2 Hierarchical construction of constant-distance groups

```

procedure ConstantDistanceCase(g)
  {g is the hierarchy of constant-distance groups}
  {g.root is the root of the dendrogram tree}
  return PostOrderTraversal(g.root);
endConstantDistanceCase

procedure PostOrderTraversal(r)

```

```

if isLeaf(r) then
  return r.name;
else
  initialize a queue Q to be empty;
  while MoreChild(r.children) do
    next = NextChild(r.children);
    add PostOrderTraversal(next) in Q;
  end while
  return Q;
end if
endPostOrderTraversal

```

Although the dendrogram can be represented as a binary tree in most cases, we use function *NextChild()* in our algorithm to deal with more general n-ary trees. Furthermore, *NextChild()* controls the order of data elements within a group. For example, we may assign higher priorities to children representing sub-groups with higher reference affinity or consider the group size or the relation with the previously traversed tree nodes. The output data hierarchy is an ordered list of data elements. If the children are traversed in the decreasing reference affinity, the output of Algorithm 2 on the Figure 1 dendrogram would be *lft*, *rtwt*, *inum*, *rt*, and *addr*. The result can be used directly to guide field re-ordering.

Our previous work shows that reference affinity is effective in improving the cache-block utilization [27]. We use data layouts obtained from the affinity analysis to guide *array regrouping* for Fortran programs and *structure splitting* for C programs. The transformation consistently outperforms other data placement methods given by the programmer, frequency-based grouping and compiler analysis across varied benchmarks and different platforms.

2.2.1 Use in Divide-and-Conquer Programs

The divide-and-conquer type of computations we consider are blocked and recursive algorithms for dense matrix operations, N-body and mesh simulation, and wavelet transform. We use N-body simulation as an example. N-body simulation considers how particles evolve in a space. The change of the speed of a particle is determined by its interaction with nearby particles within a specific radius. Mellor-Crummey et al. [17] optimized the N-body simulation problem by data and computation reordering. In one iteration of N-body simulation, most time is spent processing the interactions between nearby particles as follows. The whole space is divided into equal sized units. We order the pairs of units according to the Morton space-filling curve. An example of a Morton curve is shown in Figure 2. For any pair of units *a* and *b*, we process the interactions where the first particle appears in unit *a* and the second particle

in unit b . One can also use divide and conquer technique to implement this procedure. Take a linear array of units as an example. We recursively divide the array into two equal parts: L and R . Then the processing order is LL , LR , RL , and RR . The divide-and-conquer algorithms for matrix operations follow the same approaches of either using space-fitting curves or recursive division.

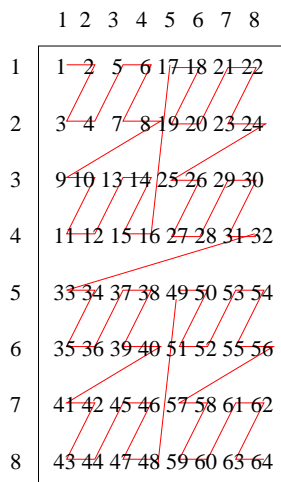


Figure 2: Morton space filling curve for an 8×8 matrix. The interactions are sorted according to their keys, which are the bit-wise interleaving of their row and column indexes.

In the following analysis, we start from one-dimensional space. Later we will extend the result to multi-dimensional space. For simplicity of analysis, we assume that the unit array is of size $n = 2^t$, where t is a non-negative integer. The N-body simulation trace is then of size 2^n . As an example, we give the trace when $n = 4$ in Figure 3. Any pair (i, j) means processing the effect of particles in unit i on particles in j .

(1,1) (1,2) (2,1) (2,2)
 (1,3) (1,4) (2,3) (2,4)
 (3,1) (3,2) (4,1) (4,2)
 (3,3) (3,4) (4,3) (4,4)

Figure 3: Trace of N-body simulation when $n = 4$

The following definition divides the unit array into equal-sized blocks of different sizes. In the following

discussion, we call a l -aligned l -sized block simply a block.

Definition 4 We call the set of units with number $i * l + 1$ to $i * l + l$ an l -aligned l -sized block, where l is a power of 2 and i is a non-negative integer.

The N-body simulation trace can be expressed using a matrix format. Figure 4 gives an illustration. In the figure, rows and columns are units and the Morton space filling curve corresponds to the trace. After the unit array is divided into blocks, the matrix is partitioned into grids. Every box, including the shaded box, is a contiguous segment of the computation trace.

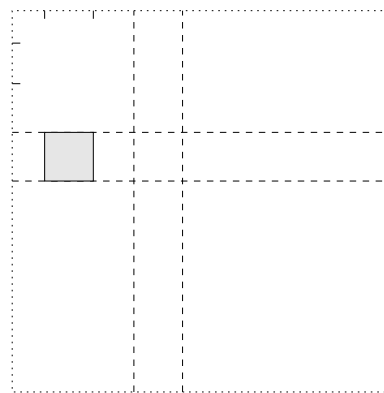


Figure 4: Illustration of trace in matrix format

For an N-body simulation trace, we have the following properties:

Lemma 1 When $k = n$, all the units belong to the same reference affinity group.

This is clear since there are at most n different units between any two units.

Next, we give a loose bound of k for a block to belong to the same k -affinity group.

Lemma 2 Every l -aligned l -sized block belongs to the same $2l$ -affinity group.

Proof Consider a box, for example, the shaded box in Figure 4. For every block, if one unit in the block appears in the box, then all of the other units appear in the box. Since there are at most $2l$ different units in every box, there is a linked path bounded by $2l$ between every two units within this box. So every l -aligned l -sized block belongs to a $2l$ -affinity group. ■

We next show that not all n units can belong to a k -affinity group for a constant k . This and the previous lemma establish that the Morton computation order has

a reference hierarchy with more than a constant number of levels.

Lemma 3 *For any k , there exists an n , such that the unit array of size n does not belong to the same k -affinity group.*

Proof We prove this by contradiction. Suppose for any n , unit array of size n belongs to the same k -affinity group. Considering the linked path between the first 1 and n , according to the definition of reference affinity, there are at most $n - 1$ segments on the path and the reuse distance of each segment is within k . Since the time distance between the first 1 and first n in the trace is of size $O(n^2)$, there must exist a segment of size at least n . However, when n is large enough, every n -sized segment of the trace has a volume distance greater than k . A contradiction. ■

Next we prove the exact structure of the reference hierarchy. Lemma 5 is the most important theoretical result. It establishes the intricate links between the linear, flexible concept of linked path and the hierarchical, rigid but complex order of recursive computation.

Lemma 4 *If one l -aligned l -sized block belongs to the same affinity group, then each l -aligned l -sized block belongs to one affinity group.*

Since all l -aligned l -sized block are symmetric in the trace, this lemma holds in a straightforward way. Next is our most important insight into the relation between two complex structures: reference affinity and recursive computation.

Lemma 5 *There exists a k , such that every l -aligned l -sized block is a separate k -affinity group, but no $2l$ -aligned $2l$ -sized block belongs to the same k -affinity group.*

Proof Let us select the k to be the smallest reuse distance that ensures an l -aligned l -sized block in the same affinity group. From lemma 2, there exists such a k . In the following proof, we suppose that $n > l$, since when $n = l$, this lemma hold trivially when $k = 1$.

Let us prove this lemma by contradiction. Suppose each $2l$ -aligned $2l$ -sized block also belongs to a k -affinity group. Without loss of generality, we analyze the first $2l$ -sized block. As shown in the upper part of Figure 5, there is a k linked path from 1 to $l + 1$. This linked path crosses 2 to l at most once. We cut this path into two parts according to the two boxes. The part in the left box starts from the first 1 to the first l . Similarly, the second part starts from the beginning to the end of the right part. Based on the symmetry of these two boxes, we can find a path for 1 to l for the first box

similar with the right part. Then we get two distinct k -linked paths from 1 to l , which are shown in the lower part of Figure 5. Circles u_1, \dots, u_s and d_1, \dots, d_t are distinct units that connect 1 and l . Actually, these units appears in the same section of trace. We put them together according to their order of the trace.

We prove that there exists a $k - 1$ linked path from 1 to l witnessed by $u_1, \dots, u_s, d_1, \dots, d_t$, where u_i and d_j denote the unit number being accessed. The lower part of Figure 5 has two paths between the same pair of end points. We show that the upper path with a link length k can be converted into a path with a link length smaller than k , which would give the desirable contradiction that the l -block is a group for a link length less than k .

We first prove for every link (u_i, u_{i+1}) , where the volume distance is k , if one d_j is located between them in the trace, then the link is divided into two links of a volume distance less than k . Consider the smallest box that includes u_i and u_{i+1} . Without loss of generality, we assume $u_i < u_{i+1}$. There are four cases to consider, as shown in Figure 6:

1. u_i and u_{i+1} are at opposite side. d_j is greater than or less than both u_i to u_{i+1} .
2. u_i and u_{i+1} are at opposite side. d_j is between u_i and u_{i+1} .
3. u_i and u_{i+1} are at the same side. d_j is greater than or less than both u_i to u_{i+1} .
4. u_i and u_{i+1} are at the same side. d_j is between u_i and u_{i+1} .

In all four cases, if d_j is in the upper half of the box, we move it to the rightmost point on the dashed line. Otherwise, move it to the leftmost point. For simplicity we explain only Case 1 and 3 for d_j greater than both u_i and u_{i+1} . The other cases are similar. The volume distance of the segment from u_i to d_j is smaller than k since the units in the lower portion are not included in this segment. The volume distance of the segment from d_j to u_{i+1} is also smaller than k , since the units smaller than d_j are not included.

To finish the conversion on the entire path, we now consider the two cases based on whether the k -distance links in the two paths overlap.

1. When the k -distance links of the two paths do not overlap. There is a d_j between u_i and u_{i+1} . This case has been just proven.
2. Otherwise, no d_j lies between u_i and u_{i+1} . Then there must exist d_j and d_{j+1} that have u_i and u_{i+1} in between. Since the volume distance between u_i and u_{i+1} is k , d_j and d_{j+1} must appear between

u_i and u_{i+1} . Considering $u_{i-1}, d_j, u_i, u_{i+1}, d_{j+1}$ and u_{i+2} : If the volume distance between u_{i-1} and d_j is smaller than k , then d_j and u_{i+1} can be divide into two links of a length smaller than k by moving u_i ; u_{i+1} and u_{i+2} can be divided into two smaller links by using d_{j+1} . Similarly we can deal with the situation when the volume distance between d_{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 $\dots, d_{j-1}, d_{j+2}, \dots$ step by step. If we can not get a k linked path at last, it means that linked path $1, u_1, \dots, u_s, l$ are exactly k -linked. This is impossible, since the original linked path goes from unit 1 to unit $l + 1$ (see the upper graph in Figure 5) is not k linked because of the last unit $l + 1$.

■

We find that the smallest k for a block of sizes 2, 4 and 6 to be in the same k -affinity group are 4, 6 and 8.

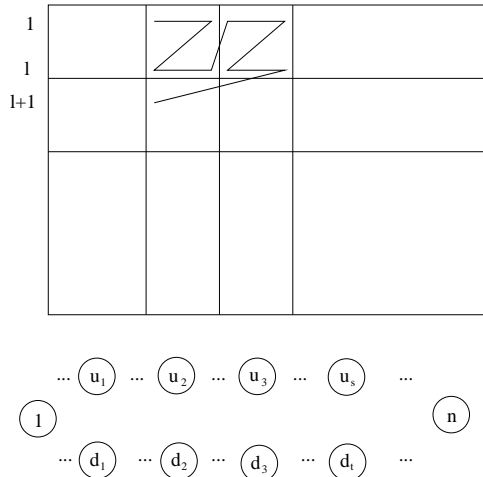


Figure 5: Illustration of the k linked paths

Following Lemmas 2 and 5, we can immediately get a hierarchical partition of the units array.

Theorem 1 *By analyzing the affinity groups of N -body simulation trace using the divide and conquer technique, we can get a hierarchical partition of the unit array with every group of a size of power of 2.*

Proof We know that when link length is 4, every 2-aligned 2-sized block is a 4-affinity group. By lemma 2 and 5, we can gradually find the lowest link length k for blocks with parameter 4, 8 \dots , every time the l -aligned l -sized blocks merge into bigger block with parameter $2l$.

■

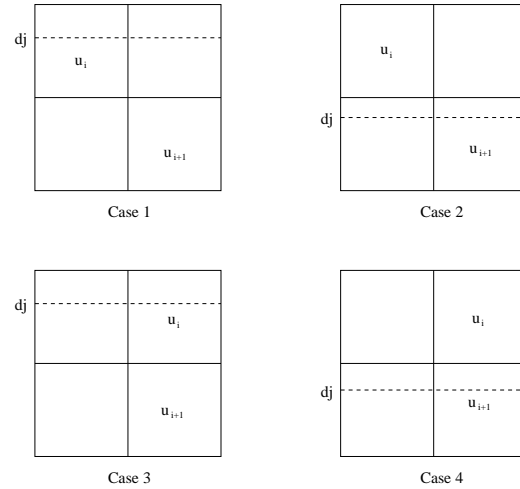


Figure 6: Four cases of places of u_i, u_{i+1} and d_j

Similarly with the previous analysis, we can get the hierarchical partition for units in 2D or 3D dimensional space. Examples are shown in figure 7.

Theorem 2 *Post-order traversal of the unit hierarchy is equivalent to the Morton space filling curve of the units in nD dimensional space.*

The proof is straightforward from Theorem 1.

2.3 Variable-Distance Groups

The link length of a variable-distance group is larger than the greatest cache size. If the group is placed in a data block, and the block is loaded into cache after the access of a group member, some members of the group may not be accessed before the block is evicted from cache. Therefore, we cannot guarantee the full block utilization as we do for constant-distance groups. For this reason, the data hierarchy method first blocks the constant-distance groups and then uses this step to increase the partial utilization. In this step, the constant-distance groups are treated as a single data element, accessed every time when its member is accessed. According to the Bounded Access Range property of reference affinity, all members are accessed within a bounded distance when one member is accessed.

Since no two elements are always accessed together within a bounded distance, we shift our attention to data groups that are often accessed together. For every pair of data elements, the following equation defines their *probable affinity* (as opposed to guaranteed affinity).

Definition 5 Probable affinity. *For any two data elements x, y , the probable affinity $p(x, y)$ is*

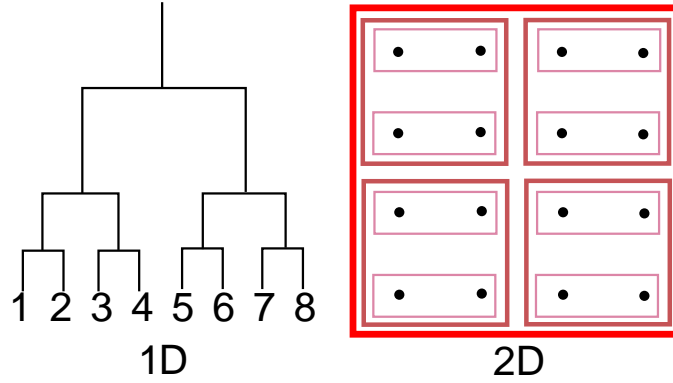


Figure 7: Hierarchical partition of the units in 1D and 2D space

$$\frac{|\{a_x \mid \exists a_y \text{ s.t. } dis(a_x, a_y) < w\}|}{|\{a_x\}| + |\{a_y\}|}$$

where w is a given window size.

The probable affinity $p(x, y)$ is the frequency of x, y being accessed together divided by the frequency of them being accessed. It is an undirected relation and $p(x, y) = p(y, x)$

Algorithm 3 shows how to construct a data hierarchy from probable affinity relations. The first step constructs an affinity graph. It searches for the smallest window size such that the graph becomes connected by edges with a probability of greater than some threshold. In later examples, we keep any edge whose probability does not converge to zero when the problem size increases. The second step iteratively builds the data hierarchy in a 3-nested repeat-until loop. The outermost loop builds level by level. The inner two loops build the next level of data blocks. It un-marks all nodes and then starts with a random node in a connected region of the graph. For each node i , it finds the neighbor j that has the most neighbors and groups j with its neighbors, including i . It uses a work list to cover the entire region. It uses marks to ensure that each node is grouped only once at each level. Eventually all connected regions of the graph becomes a hierarchy of node blocks. To make this algorithm more efficient, we can search for only the power-of-two window sizes.

Algorithm 3 Hierarchical construction of variable-distance groups

procedure *VariableDistanceCase*(g)
 $\{g$ is the hierarchy of variable-distance groups}

Collapse constant-distance groups as single data elements, accessed whenever a group member is accessed

{Constructing the probable affinity graph}
for window size $w = 2$ to the largest cache size **do**
 Construct the probable affinity graph G
 if G is fully connected **then**
 Exit the loop
 end if
end for
{Constructing the data hierarchy}
repeat
 Clear the mark on all elements in G
 Initialize a queue Q to be empty
 repeat
 Place a random, unmarked data element in Q
 repeat
 Dequeue the top element i from Q
 Find its unmarked neighbor j that has the most unmarked neighbors
 Group j and all its unmarked neighbors
 Collapse the group as a single element connected to the neighbors of the group members
 Mark the group
 Place the immediate, unmarked neighbors into Q
 until Q is empty
 until all elements are marked
until the graph consists of single nodes
return the data hierarchy
end *VariableDistanceCase*

Algorithm 3 is heuristic based and does not guarantee an optimal placement. It uses pair-wise frequency,

which has been studied in the past [4, 7, 12, 22]. We can adjust the parameters—considering the size of the data element and changing the window size and the cut-off threshold—or fine tune the relation definition—counting only the accesses that are separated by some volume distance. While we may improve the current method using these extensions or other techniques from previous studies, the unique feature of the algorithm is not the probability affinity graph but the bottom-up construction of a data hierarchy. It starts with the immediate neighbors and gradually adds levels to the hierarchy. Unlike previous work in pair-wise affinity, we do not pack for a fixed-size data block. Instead we use the natural graph topology for a hierarchical construction. In addition, Algorithm 3 is used *only* after we have exploited constant-distance affinity groups.

Next we discuss three example uses of Algorithm 3. In each case, it in fact automatically constructs the best data hierarchy.

Random Access Sequences As an extreme case, consider a group of n data being repeatedly accessed in a random order. The affinity hierarchy has two levels. When k is 0, each element is a group. When k is $n - 1$, all elements are in a single group. Hence, our base algorithm would invoke Algorithm 3. For any constant window size, the probability graph has no edges because the probability of any two data element accessed together is $1/n$, which is arbitrarily close to zero when n is sufficiently large. Therefore, the construction part finds no possibility of grouping any data elements. The data hierarchy has a single level, where each element is a block. This result is correct because any data layout is equally good for a random access sequence.

Binary Search Trees Consider a balanced binary search tree of n nodes. Assume that the access is a sequence of random queries of the leaf nodes. We omit the proof here, but the affinity hierarchy is as follows. When k is a constant, there is no non-trivial affinity groups. When k is $\frac{n+1}{2^k} - 1$, the top $k + 1$ layers of the tree is only non-trivial affinity group at that k . Therefore, Algorithm 3 is invoked. If it picks the window size to be 2, the probable affinity graph is the same as the tree. The hierarchical grouping would produce the first and second level blocks as shown in Figure 8. Every first level block has four tree nodes. Every second level block has six first level blocks. The data layout has the same recursive structure as the van Emde Boas layout for binary search trees, which has been shown to be “cache-oblivious” by Bender et al [2]. Their method divides the tree top down with three tree nodes in the lowest level block. The different shape of the block does not affect the asymptotic optimality. In fact, the size of higher

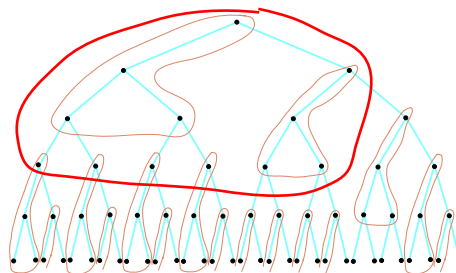


Figure 8: Recursive blocking of a balanced binary tree under random searches

level blocks increases in the same exponential rate in both methods.

Our method has two advantages over cache-oblivious search trees. First, we build the hierarchy bottom up, so we do not need to require the tree to be balanced. Second, the van Emde Boas layout is optimal if the tree is searched randomly. Random access is only one of the many possible tree-search patterns. Our method adapts data layouts with the access pattern. For example, if some tree nodes are always searched together, our algorithm will exploit the reference affinity first and give better cache performance than the van Emde Boas layout.

Random Walks As an exercise, we try our algorithm on a new data placement problem not yet studied in the literature. Given a grid and a starting point, a random walk starts from the point and at this and succeeding points has equal probabilities to walk to one of the neighboring point. For a two-dimensional square grid of size n , the affinity hierarchy has no non-trivial constant-distance affinity groups. Algorithm 3 picks the window size to be 2. The probable affinity graph is the same as the grid. The algorithm will construct a hierarchy as shown in Figure 9. The first and other level blocks exactly captures the locality in a random walk.

Random walks have many uses in computer science as well as other fields such as economics. A common example is the large hidden Markov models used in natural language processing and computer vision. This result suggest that our algorithm can automatically identify the structure of transition network, construct the data hierarchy, and improve the cache performance of these applications.

2.4 Time Complexity

We make an approximate complexity analysis here since the exact affinity analysis and the input form of the affinity hierarchy is not fixed. For the same reason we do not discuss the space complexity. Algorithm 1 iterates all affinity hierarchy levels, which is at most n , where

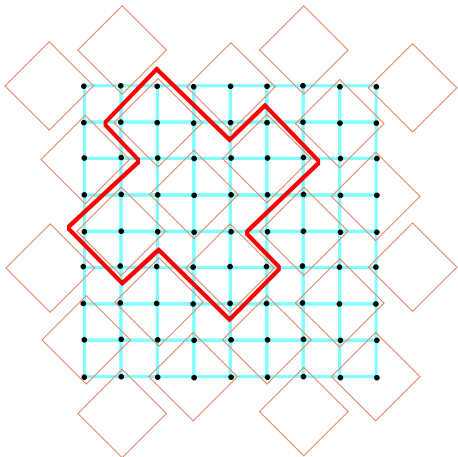


Figure 9: Recursive blocking of a grid under a random walk

n is the size of program data. Assuming the identity check between two levels is constant time, the cost is $O(n)$. Algorithm 2 uses a post-order traversal of a constant number of layers of the affinity hierarchy. The cost is again $O(n)$. Algorithm 3 uses $O(t)$ to construct an probable affinity graph, where t is the length of the trace. It finds connected components in $O(n^2)$. The recursive construction takes $O(n \log n)$. The total time excluding affinity analysis is therefore $O(t + n^2)$. It can be improved by a sampling method to construct the probably affinity graph [25] and by bounding the degree of node in a graph.

The affinity analysis uses efficient heuristics including k -distance clustering [27] and sampling [25]. The first method takes $O(n^2)$ and needs the measurement of reuse distance for every element of the trace. Reuse distance can be approximated by a precision arbitrarily close to 100% in $O(t \log \log n)$ time, which is effectively linear time for any real data size [9].

3 Related Work

Chatterjee et al. demonstrated the performance advantage of the hierarchical data layout for matrix multiply, Cholesky factorization, and wavelet transform [5]. Mellor-Crummey et al. studied the blocked sparse data layout for irregular programs such as N-body and mesh simulation [17]. The blocked data placement in these applications is mostly manual and not automated. An exception is the consecutive packing by Ding and Kennedy [8], which is shown in practice to be a good approximation [17, 21]. Bender et al. studied the recursive van Emde Boas layout for search trees, designed algorithms for a dynamic B-tree, and proved the optimal cost for random searches [2]. Like the first two studies, the data layout is designed manually.

The method in this paper is the first to give the equivalent hierarchical data layout for application domains as important and diverse as linear systems, N-body simulation, and databases. While it does not displace the research for finding and proving the best data layout, our automatic method assists programming and implementation. One remaining implementation issue is the placement order of the sub-blocks within a block. Previous results suggest that the best order is the one with the least overhead [17].

Early forms of compiler analysis identify groups of data that are used together in loop nests [16, 22, 23]. Analysis for general-purpose programs is often based on the profiling of the access to data elements, data pairs, and data streams [4, 6, 7, 12, 14, 20, 22]. Although effective in improving program locality, none of these models are general enough to give the best layout for any of the cases we described in the paper. Reuse distance combines frequency information with the time and the volume of data access. Experimental results have shown that reuse distance models are better in data reorganization and locality prediction [3, 9, 10, 15, 26, 27]. In this paper, we build on the past reuse distance and reference affinity models and show a general method for hierarchical data placement.

Thabit showed that data packing for a given block size is NP-hard. Kennedy and Kremer gave a general model that considered run-time data transformation (among other techniques) and showed that the problem is NP-hard [13]. PetrankRawitz:Hardness Petrank and Rawitz showed that if $P \neq NP$, no polynomial time method can guarantee a data layout whose number of cache misses is within $O(n^{1-\epsilon})$ of that of the optimal data layout, where n is the data size. It further shows that using only pair-wise information, no algorithm can guarantee a data layout whose number of cache misses is within $O(k-3)$ of that of the optimal data layout, where k is the size of cache. The harsh bound is true if we consider all programs. However, many programs have consistent data patterns that can be exploited. In particular, with reference affinity, our method can guarantee the full utilization of the data block in general cases and can find best data layout in diverse and important special cases. When the analytical method is inadequate, one can search the solution space using an efficient feedback and evaluation system [19].

Loop tiling has long been studied for improving the data reuse in registers, single- or multi-level cache. The transformations and their basis in dependence theory are described by many authors (see [1] for a recent example). Divide-and-conquer is a common algorithm framework, known for its efficiency for a storage hierarchy. Recursive algorithms such as matrix multiply has a long history, so are the use of space-fitting curves in com-

putation reordering in diverse application domains, as discussed in the literature (see [5] and [11] for examples). Leisersen’s group at MIT proved the asymptotic optimality on a realistic cache model and called such algorithms *cache oblivious* [11]. While most recursive programs are written manually, they can be generated automatically by a compiler from an iterative form [24]. This work is also related to the research in cache and memory models, which are the basis for the optimality results. This work is complementary. It identifies the data locality and automates the data placement once the computation order is optimized.

While optimization of general programs is most often NP-hard and poorly approximable, the research community has been successful in finding sub-classes of programs that can be optimized or improved. This paper makes another step in this direction by presenting a general model and method for hierarchical data placement.

4 Summary

In this paper we have presented till now the most general method for hierarchical data placement. Based on the reference affinity, the method divides the problem into two cases. It uses a tree traversal for constant-distance groups and a bottom-up construction for variable-distance groups. It automatically gives the desirable data hierarchy not only for the diverse cases reported by past studies—matrix multiplication, Cholesky factorization, wavelet transform, N-body simulation, sparse mesh, and search trees—but also for important new cases—random accesses and random walks. The theorems and proofs have established the subtle link between the path-based affinity models and the hierarchical structure of the computation. These results advance our understanding of the hierarchical data placement and open the door for research in more effective and efficient placement methods for general-purpose programs. The strong relation between the access pattern in computation and the spatial relation in data gives us a basis for improving the programming, compiler, and language support for modern computer memory systems.

Acknowledgments

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

References

- [1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 2001.
- [2] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *Proceedings of Symposium on Foundations of Computer Science*, November 2000.
- [3] K. Beyls and E. D’Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, Paderborn, Germany, August 2002.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] C. Fang, S. Carr, S. Onder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the first ACM*

- SIGPLAN Workshop on Memory System Performance*, Washington DC, June 2004.
- [11] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of Symposium on Foundations of Computer Science*, October 1999.
- [12] N. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5), September 1999.
- [13] K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4), 1998.
- [14] D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.
- [15] 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.
- [16] 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.
- [17] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming*, 29(3), June 2001.
- [18] 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.
- [19] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data layout optimizations. In *Proceedings of ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.
- [20] 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.
- [21] 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.
- [22] K. O. Thabit. *Cache Management by the Compiler*. PhD thesis, Dept. of Computer Science, Rice University, 1981.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] 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.