

# Lightweight Reference Affinity Analysis

Xipen Shen\*, Yaoqing Gao+, Chen Ding\*, and Roch Archambault+

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

+ IBM Toronto Software Lab  
Toronto, ON, L6G 1C7, Canada  
{ygao,archie}@ca.ibm.com

## ABSTRACT

Previous studies have shown that array regrouping and structure splitting significantly improve data locality. The most effective technique relies on profiling every access to every data element. The high overhead impedes its adoption in a general compiler. In this paper, we show that for array regrouping in scientific programs, the overhead is not needed since the same benefit can be obtained by pure program analysis.

We present an interprocedural analysis technique for array regrouping. For each global array, the analysis summarizes the access pattern by access-frequency vectors and then groups arrays with similar vectors. The analysis is context sensitive, so it tracks the exact array access. For each loop or function call, it uses two methods to estimate the frequency of the execution. The first is symbolic analysis in the compiler. The second is lightweight profiling of the code. The same interprocedural analysis is used to cumulate the overall execution frequency by considering the calling context. We implemented a prototype of both the compiler and the profiling analysis in the IBM® compiler, evaluated array regrouping on the entire set of SPEC CPU2000 FORTRAN benchmarks, and compared different analysis methods. The pure compiler-based array regrouping improves the performance for the majority of programs, leaving little room for improvement by code or data profiling.

## Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors – *compilers and optimization*.

## Keywords

Affinity, Frequency, Compiler, Data Regrouping, Data Interleaving, Memory Optimization

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'05, June 20-22, Boston, MA, USA.

Copyright 2005, ACM 1-59593-167-8/06/2005...\$5.00

## 1. INTRODUCTION

Over the past 30 years, memory performance increasingly determines the program performance on high-end machines. Although programs employ a large amount of data, they do not use all data at all times. We can improve cache spatial locality by storing in cache precisely the data that is required at a given point of computation. In scientific programs, most data is stored in arrays. In this paper, we study the organization of data in multiple arrays.

Figure 1 shows an example of array regrouping. Part (a) shows a program that uses four attributes of  $N$  molecules in two loops. One attribute, “position”, is used in both the compute loop and the visualization loop, but the other three are used only in the compute loop. Part (b) shows the initial data layout, where each attribute is stored in a separate array. In the compute loop, the four attributes of a molecule are used together, but they are stored far apart in memory. On today's high-end machines from IBM, Microsystems, and companies using Intel® Itanium® and AMD processors, the largest cache in the hierarchy is composed of blocks of no smaller than 64 bytes. In the worst case, only one 4-byte attribute is useful in each cache block, 94% of cache space would be occupied by useless data, and only 6% of cache is available for data reuse. A similar issue exists for memory pages, except that the utilization problem can be much worse.

Array regrouping improves spatial locality by grouping three of the four attributes together in memory, as shown in part (c) of Figure 1. After regrouping, a cache block should have at least three useful attributes. One may suggest grouping all four attributes. However, three of the attributes are not used in the visualization loop, and therefore grouping them with “position” hurts cache-block utilization. However, if the loop is infrequently executed or it touches only a few molecules, then we may still benefit from grouping all four attributes.

Array regrouping has many other benefits. First, it reduces the interference among cache blocks because fewer cache blocks are accessed. By combining multiple arrays, array regrouping reduces the page-table working set and consequently the number of Translation Lookaside Buffer (TLB) misses in a large program. It also reduces the register pressure because fewer registers are needed to store array base addresses. It may improve energy efficiency by allowing more memory pages to enter a sleeping model. For the above reasons, array regrouping is beneficial even for arrays that are contiguously accessed.

```

% attributes for molecules
position[N], speed[N], mass[N]
force[N]
... ..
compute_loop
  position[m]=f(position[m], speed[m],
                force[m], mass[m])
... ..
visualization_loop
  display(position[k])
... ..
(a) a program that uses four arrays

```



**Figure 1. Array regrouping example. Data with reference affinity is placed together to improve cache utilization**

These benefits have been verified in our previous study [6]. Finally, on shared-memory parallel machines, better cache-block utilization means slower amortized communication latency and better bandwidth utilization.

Array regrouping is mostly orthogonal to traditional loop-nest transformations and single-array transformations. The latter two try to effect contiguous access within a single array. Array regrouping complements them by exploiting cross-array spatial locality, even when per-array data access is contiguous. As a data transformation, it is applicable to irregular programs where the dependence information is lacking. In the example in Figure 1, the correctness of the transformation does not depend on knowing the value of index variables  $m$  and  $k$ . While array regrouping has a good potential for complex programs, it has not been implemented in any production compiler because the current techniques are not up to the task.

Ding and Kennedy gave the first compiler technique for array regrouping [6]. They defined the concept reference affinity. A group of arrays have reference affinity if they are always accessed together in a program. Their technique is conservative and groups arrays only when they are always accessed together. We call this scheme conservative affinity analysis. Conservative analysis is too restrictive in real-size applications, where many arrays are only sporadically accessed.

Zhong et al. redefined reference affinity at the trace level using a concept called reuse distance, which is the volume of data between two accesses of the same unit of data. They grouped arrays that have similar distributions of reuse distances (reuse signatures) [15]. We call it distance-based affinity analysis. The new scheme groups arrays if they are mostly used together and outperforms the conservative scheme for a set of three

FORTRAN programs. Reuse-distance profiling, however, carries a high overhead. The slowdown is at least 10 to 100 times. No production compiler is shipped with such a costly technique. No one would do so before carefully examining whether such a high cost is justified.

The two previous techniques were evaluated on a small set of programs, partly because the techniques did not handle parameter arrays as well as global arrays that are passed as parameters. Since multiple arrays may map to the same parameter array at different times, the affinity information is ambiguous. Another problem is aliasing, which has not been considered in array regrouping.

We present frequency-based affinity analysis. It uses a frequency-based model to group arrays even if they are not always accessed together. It uses interprocedural program analysis to measure the access frequency in the presence of array parameters and aliases. To collect the frequency within a loop or a function, we study two methods. The first is symbolic analysis by a compiler. The second is lightweight profiling. The techniques apply to FORTRAN programs. In the rest of the paper, we will present the frequency model, the estimation methods, and the interprocedural analysis. We will describe their implementation in the IBM® FORTRAN compiler, an evaluation on SPEC CPU2000 floating-point benchmark programs, and comparisons between frequency-based and distance-based methods, and between pure compiler analysis and lightweight profiling.

## 2. FREQUENCY-BASED AFFINITY ANALYSIS

Below is the general framework of the analysis.

- Building the control flow graph and the invocation graph with data flow analysis
- Estimating the execution frequency through either static analysis or profiling
- Building array access-frequency vectors using interprocedural analysis, as shown in Figure 2.
- Calculating the affinity between each array pair and constructing the affinity graph
- Partitioning the graph to find affinity groups in linear time

In this section, we first present the affinity model, where arrays are nodes and affinities are edge weights in the affinity graph, and the affinity groups are obtained through linear-time graph partitioning. We then describe the two methods, static and lightweight profiling, for collecting the frequency information. Finally, we describe the context-sensitive interprocedural reference affinity analysis and the use of the frequency information by the analysis.

### 2.1 Frequency-based Affinity Model

A program is modeled as a set of code units, in particular, loops. Suppose there are  $K$  code units. Let  $f_i$  represent the total occurrences of the  $i$ th unit in the program execution. We use  $r_i(A)$  to represent the number of references to array  $A$  in an execution of the  $i$ th unit. The frequency vector of array  $A$  is defined as follows:

$$V(A) = (v_1, v_2, \dots, v_K)$$

where

$$\begin{aligned} v_i &= 0 & \text{if } r_i(A) = 0; \\ v_i &= f_i & \text{if } r_i(A) > 0. \end{aligned}$$

A code unit  $i$  may have branches inside and may call other functions. We conservatively assume that a branch goes both directions when collecting the data access. We use interprocedural analysis to find the side effects of function calls.

To save space, we can use a bit vector to replace the access vector of each array and use a separate vector to record the frequency of code units.

The affinity between two arrays is the Manhattan distance between their access-frequency vectors, as shown below. It is a number between zero and one. Zero means that two arrays are never used together, while one means that both are accessed whenever one is.

$$affinity(A, B) = 1 - \frac{\sum_{i=1}^K |v_i(A) - v_i(B)|}{\sum_{i=1}^K (v_i(A) + v_i(B))}$$

We construct an affinity graph. Each node represents an array, and the weight of an edge between two nodes is the calculated affinity between them. There are additional constraints. To be regrouped, two arrays must be compatible in that they should have the same number of elements and they should be accessed in the same order [6]. The data access order is not always possible to analyze at compile time. However, when the information is available to show that two arrays are not accessed in the same order in a code unit, the weight of their affinity edge will be reset to zero. The same is true if two arrays differ in size.

Graph partitioning is done through a graph traversal. It merges two nodes into a group if the affinity weight is over a threshold. After partitioning, each remaining node is a set of arrays to be grouped together. The threshold determines the minimal amount of affinity for array regrouping. We will examine the effect of different thresholds in Section 4. The entire algorithm, including graph partitioning, is given in Figure 2.

### 2.2 Unit of Program Analysis

For scientific programs, most data accesses happen in loops. We use a loop as a hot code unit for frequency counting for three reasons: coverage, independence, and efficiency.

- Coverage: A loop often accesses an entire array or most of an array. In that case, branches and function calls outside the loop have no effect on whether two arrays are accessed together or not.
- Independence: McKinley and Temam reported that most cache misses in SPEC95 FP programs were due to cross-loop reuses [16]. We expect the same for our test programs and ignore the cache reuse across two loops. Therefore, the temporal order in which loops are executed has no effect on the affinity relation. Without the independence, when two arrays appear in different code units, their affinity may depend on the temporal relations across units. The independence property simplifies the affinity analysis by allowing it to compose the final result from analyzing individual code units.
- Efficiency: The total number of loops determines the size of the access-frequency vector. In a context-sensitive analysis, a unit becomes multiple elements in the access-frequency vector, one for each distinct calling context. The number of loops is small enough to enable full context-sensitive analysis, as described in Section 2.5. In our experiment, the maximum is 351 for benchmark *Galgel*.

In comparison, other types of code units are not as good for array regrouping. For example, a basic block has too little data access to be independent from other basic blocks. Basic blocks may be too numerous for compiler analysis or lightweight profiling to be affordable. A small procedure lacks independence in data access. A large procedure has less coverage because it often has a more complex control flow than a loop does. Other possible code units are super-blocks and regions, but none satisfies the three requirements as well as

loops do. Loops have good independence, so the temporal order of loops has little impact on the affinity result. The number of loops is not overly large in most programs. Branches inside loops hurt the coverage. However, very few branches exist in loops in scientific programs, especially in the innermost loop.

## 2.3 Static Estimate of the Execution Frequency

Many past studies have developed compiler-based estimate of the execution frequency (e.g., [11, 13]). The main difficulties are to estimate the value of a variable, to predict the outcome of a branch, and to cumulate the result for every statement in a program. We use standard constant propagation and symbolic analysis to find constants and relations between symbolic variables.

We classify loops into three categories. The bounds of the first group are known constants. The second group of loops have symbolic bounds that depend on the input, e.g. the size of the grid in a program simulating a three-dimensional space. The number of iterations can be represented by an expression of a mix of constants and symbolic values. We need to convert a symbolic expression into a number because the later affinity analysis is based on numerical values. The exact iteration count is impossible to obtain. To distinguish between high-trip count loops from low-trip count loops, we assume that a symbolic value is reasonably large (100) since most low-trip count loops have a constant bound. This strategy works well in our experiments.

The third category includes many while-loops, where the exit condition is calculated in each iteration. Many while-loops are small and do not access arrays, so they are ignored in our analysis. In other small while-loops, we take the size of the largest array referenced in the loop as the number of iterations. If the size of all arrays is unknown, we simply assign a constant 100 as the iteration count.

The array regrouping is not very sensitive to the accuracy of loop iteration estimations. If two arrays are always accessed together, they would be regarded as arrays with perfect affinity regardless how inaccurate the iteration estimations are. Even for arrays without perfect affinity, the high regrouping threshold provides good tolerance of estimation errors as discussed in Section 4.1.

The frequency of the innermost loop is the product of its iteration count, the number of iterations in all enclosing loops in the same procedure, and the estimated frequency of the procedure invocation. The execution frequency of loops and subroutines is estimated using the same interprocedural analysis method described in Section 2.5. It roughly corresponds to inlining all procedural calls.

For branches, we assume that both paths are taken except when one branch leads to the termination of a program, i.e., the stop statement. In that case, we assume that the program does not follow the exit branch. This scheme may overestimate the affinity relation. Consider a loop whose body is a statement

with two branches  $\alpha$  and  $\beta$ . Suppose array A is accessed in the branch  $\alpha$  and B in the  $\beta$  branch. In an execution, if the two branches are taken in alternative loop iterations, then the affinity relation is accurate, that is, the two arrays are used together. However, if  $\alpha$  is taken in the first half iterations and  $\beta$  in the second half (or vice versa), then the two arrays are not used together. The static result is an overestimate.

## 2.4 Profiling-based Frequency Analysis

By instrumenting a program, the exact number of iterations becomes known for the particular input. To consider the effect of the entire control flow, we count the frequency of execution of all basic blocks. Simple counting would insert a counter and an increment instruction for each basic block. In this work, we use the existing implementation in the IBM compiler [12], which implements more efficient counting by calculating from the frequency of neighboring blocks, considering a flow path, and lifting the counter outside a loop. Its overhead is less than 100% for all programs we tested. The execution frequency for an innermost loop is the frequency of the loop header block. When a loop contains branches, the analysis is an overestimate for reasons described in Section 2.3.

## 2.5 Context-sensitive Interprocedural Reference Affinity Analysis

Aliases in FORTRAN programs are caused by parameter passing and storage association. We consider only the first cause. We use an interprocedural analysis based on the invocation graph, as described by Emami et al [7]. Given a program, the invocation graph is built by a depth-first traversal of the call structure starting from the program entry. Recursive call sequences are truncated when the same procedure is called again. In the absence of recursion, the invocation graph enumerates all calling contexts for an invocation of a procedure. A special back edge is added in the case of a recursive call, and the calling context can be approximated.

The affinity analysis proceeds in two steps. The first step takes one procedure at a time, treats the parameter arrays as independent arrays, identifies loops inside the procedure, and the access vector for each array. The procedure is given by *BuildStaticAFVList* in Figure 2.

The second step traverses the invocation graph from the bottom up. At each call site, the affinity results of the callee are mapped up to the caller based on the parameter bindings, as given by procedures *BuildDynamicAFVList*, *UpdateAFVList*, and *UpdateDyn* in Figure 2. As an implementation, the lists from all procedures are merged in one vector, and individual lists are extracted when needed, as in *UpdateDyn*. The parameter binding for a recursive call is not always precise. But a fixed point can be obtained in linear time using an algorithm proposed by Cooper and Kennedy (Section 11.2.3 of [1]).

Because of the context sensitivity, a loop contributes multiple elements to the access-frequency vector, one for every calling context. However, the number of calling contexts is small. Emami et al. reported on average 1.45 invocation nodes per call

site for a set of C programs. [7]. We saw a similar small ratio in FORTRAN programs.

The calculation of the access-frequency vector uses the execution frequency of each loop, as in procedure *UpdateDyn*. In the case of static analysis, the frequency of each invocation node is determined by all the loops in its calling context, not including the back edges added for recursive calls. The frequency information is calculated from the top down. Indeed, in our implementation, the static frequency is calculated at the same time as the invocation graph is constructed.

The frequency from the lightweight profiling can be directly used if the profiling is context sensitive. Otherwise, the average is calculated for the number of loop executions within each function invocation. The average frequency is an approximation.

The last major problem in interprocedural array regrouping is the consistency of data layout for parameter arrays. Take, for example, a procedure that has two formal parameter arrays. It is called from two call sites; each passes a different pair of actual parameter arrays. Suppose that one pair has reference affinity but the other does not. To allow array regrouping, we will need two different layouts for the formal parameter arrays. One possible solution is procedural cloning, but this leads to code expansion, which can be impractical in the worst case. In this work, we use a conservative solution. The analysis detects conflicts in parameter layouts and disables array regrouping to resolve a conflict. In the example just mentioned, any pair of arrays that can be passed into the procedure are not regrouped. In other words, array regrouping guarantees no need of code replication in the program.

The invocation graph excludes pointer-based control flow and some use of dynamically loaded libraries. The former does not exist in FORTRAN programs and the latter is a limitation of static analysis.

### 3. IMPLEMENTATION

This work is implemented in IBM® TPO (Toronto Portable Optimizer), which is the core optimization component in IBM® C/C++ and FORTRAN compilers. It implements both compile-time and link-time methods for intra- and interprocedural optimizations. It also implements profiling feedback optimizations. We now describe the structure of TPO and the implementation of the reference affinity analysis.

TPO uses a common graph structure based on Single Static Assignment form (SSA) [1] to represent the control and data flow within a procedure. Global value numbering and aggressive copy propagation are used to perform symbolic analysis and expression simplifications. It performs pointer analysis and constant propagation using the same basic algorithm from Wegman and Zadeck [14], which is well suited for using SSA form of data flow. For loop nests, TPO performs data dependence analysis and loop transformations after data flow optimizations. We use symbolic analysis to identify the bounds of arrays and estimate the execution frequency of loops. We use dependence analysis to identify regular access patterns to arrays.

During the link step, TPO is invoked to re-optimize the program. Having access to the intermediate code for all the procedures in the program, TPO can significantly improve the precision of the data aliasing and function aliasing information. Interprocedural mod-use information is computed at various stages during the link step.

The reference affinity analysis is implemented at the link step. A software engineering problem is whether to insert it before or after loop transformations. Currently the analysis happens first, so arrays can be transformed at the same compilation pass as loops are. As shown later, early analysis does not lead to slower performance in any of the test programs. We are looking at implementation options that may allow a later analysis when the loop access order is fully determined.

We have implemented the analysis that collects the static access-frequency vector and the analysis that measures per-basic-block execution frequency through profiling. We have implemented a compiler flag that triggers either static or profiling-based affinity analysis. The invocation graph is part of the TPO data structure. We are in the process of completing the analysis that includes the complete context sensitivity. The current access-frequency vector takes the union of all contexts. We have implemented the reference affinity graph and the linear-time partitioning. The array transformations are semi-automated as the implementation needs time to fully bond inside the compiler.

The link step of TPO performs two passes. The first is a forward pass to accumulate and propagate constant and pointer information within the entire program. Reference affinity analysis is part of the global reference analysis used for remapping global data structures. It can clone a procedure [1] when needed, although we do not use cloning for array regrouping. The second pass traverses the invocation graph backward to perform various loop transformations. Interprocedural code motion is also performed during the backward pass. This transformation will move upward from a procedure to all of its call points. Data remapping transformations, including array regrouping when fully implemented, are performed just before the backward pass to finalize the data layout. Loop transformations are performed during the backward pass to take full advantage of the interprocedural information. Interprocedural mod-use information is recomputed again in order to provide more accurate information to the back-end code generator.

### Data Structure

staticAFV List : the list of local access-frequency vectors, one for each array and each subroutine

dynAFV List : the list of global access-frequency vectors, one for each array

loopFreq : the local estimate of the execution frequency of a loop

IGNode : the data structure of a node in the invocation graph, with the following attributes

freq : the estimated frequency of the node

staticStartId : the position of the subroutine's first loop in staticAFVList vectors

dynStartId : the position of the subroutine's first loop in dynAFVList vectors

groupList : the list of affinity groups

### Algorithm

- 1) building control flow graph and invocation graph with data flow analysis
- 2) estimating the execution frequency through either static analysis or profiling (Section 2.3 and 2.4)
- 3) building array access-frequency vectors using interprocedural analysis (this algorithm, explained in Section 2.5)
- 4) calculating the affinity between each array pair and constructing the affinity graph (Section 2.1)
- 5) linear-time graph partitioning to find affinity groups (Section 2.1)

#### Procedure BuildAFVList()

// build access frequency vectors

BuildStaticAFVList ();

BuildDynamicAFVList ();

End

#### Procedure BuildStaticAFVList()

// local access frequency vectors

id = 0;

For each procedure proc

For each inner-most loop l in proc

refSet = GetArrayRefSet(l);

If (refSet == NULL)

Continue;

End

id ++;

For each member a in refSet

staticAFVList[a][id]=loopFreq(l);

End

End

End

End

#### Procedure BuildDynamicAFVList()

// global access frequency vectors

For each leaf node n in the invocation graph

UpdateAFVList(n);

End

End

#### Procedure UpdateAFVList(IGNode n)

For each array a in n.refSet

UpdateDyn(a,n);

End

par = n.Parent();

If (par == NULL)

return;

End

For each array virtual parameter p

q = GetRealParameter(p);

UpdateDyn(q,n);

End

n.visited = true;

If (IsAllChildrenUpdated(par))

UpdateAFVList(par);

End

End

#### Procedure UpdateDyn(array a, IGNode n)

s1=n.staticStartId;

s2=n.dynStartId;

i=0;

While(i<n.loopNum)

dynAFVList[a][s2+i] +=staticAFVList[a][s1+i]\*n.freq;

i++;

End

End

#### Procedure GraphPartition()

// partition into affinity groups

For each edge e in the affinity graph g

If (edge.affinity > Threshold)

g.merge(edge);

End

End

groupList = g.GetNodeSets();

End

Figure 2: Interprocedural reference affinity analysis

## 4. EVALUATION

We test on two machine architectures shown in Table 1. We use 11 benchmarks for testing. Eight are from SPEC CPU2000. The other three are programs used in distance-based affinity analysis by Zhong et al. [15]. Table 2 gives the source and a description of the test programs. Most of them are scientific simulations for quantum physics, meteorology, fluid and molecular dynamics. Two are image processing and number theory. The table shows that they use from 4 to 92 arrays.

**Table 1: Machine architectures**

Machine Type	IBM® p690 Turbo+	Intel® PC
Processor	IBM® POWER4™+ 1.7GHz	Pentium® 4 2.8GHz
L1 data cache	32 KB, 2-way, 128 B cache line	8 KB, 64 B cache line
L2 data cache	1.5 MB, 4-way	512 KB, 8-way

**Table 2: Test programs**

Benchmark	Source	Description	Arrays
Applu	Spec2K	Physics/Quantum Chromodynamics	38
Apsi	Spec2K	Meteorology:Pollutant Distribution	92
Facerec	Spec2K	Image Processing: Face Recognition	44
Galgel	Spec2K	Computational Fluid Dynamics	75
Lucas	Spec2K	Number Theory/Primality Testing	14
Mgrid	Spec2K	Multi-grid Solver:3D Potential Field	12
Swim2K	Spec2K	Shallow Water Modeling	14
Wupwise	Spec2K	Physics/Quantum Chromodynamics	20
Swim95	Zhong+	Shallow Water Modeling	14
Tomcatv	Zhong+	Vectorized Mesh Generation	9
MolDyn	Zhong+	Molecular Dynamics Simulation	4

### Affinity groups

Table 3 shows the affinity groups identified by interprocedural reference affinity analysis using static estimates. The program that has most non-trivial affinity groups is *Galgel*. It has eight affinity groups, including 24 out of 75 arrays in the program. Four programs---*Apsi*, *Lucas*, *Wupwise*, and *MolDyn*---do not have affinity groups with more than one array. *Apsi* uses only one major array, although parts of it are taken as many arrays in over 90 subroutines. It is possible to split the main array into many smaller pieces. It remains our future work. *Lucas*, *Wupwise*, and *MolDyn* have multiple arrays but no two have strong reference affinity. The affinity groups in *Facerec* and *Mgrid* contain only small arrays. The other three SPEC CPU2000 programs, *Applu*, *Galgel*, and *Swim2K*, have reference affinity among large arrays.

**Table 3: Affinity groups**

Benchmark	Affinity groups
Applu	(imax,jmax,kmax) (idmax,jdmax,kdmax) (phi1,phi2) (a,b,c) (ldx,ldy,ldz) (udx,udy,udz)
Apsi	<none>
Facerec	(coordx,coordy)
Galgel	(g1,g2,g3,g4) (f1,f2,f3,f4) (vyy,vyy2,vxy,vxy2) (vxxx,vyxx) (vyyy,vxxy,vxyy,vyxy) (v1,v2) (wxtx,wytx) (wypy,wxpy)
Lucas	<none>
Mgrid	(j1,j2,j3)
Swim2K	(unew,vnew,pnew) (u,v) (uold,vold,pold) (cu,cv,z,h)
Wupwise	<none>
Swim95	(unew,vnew,pnew) (u,v) (uold,vold,pold) (cu,cv,z,h)
	compared to [15]: (unew,vnew,pnew) (u,v) (uold,pold) (vold) (cu,cv,z,h)
Tomcatv	(x,y) (rxm,rym) (rx,ry)
	compared to [15]: (x,y) (rxm,rym) (rx,ry)
MolDyn	<none>
	compared to [15]: <none>

### Comparison with distance-based affinity analysis

*Swim95*, *Tomcatv*, and *MolDyn* are all FORTRAN programs tested by Zhong et al. [15]. Their distance-based analysis measures the reuse distance of every access in a trace. The profiling time is in hours for a program. Zhong et al. also used the profiling method for structure splitting in C programs. We consider only FORTRAN programs in this study.

The bottom six rows of Table 3 compare the affinity groups from reuse-distance profiling. Our program analysis gives the same results for *Tomcatv* and *MolDyn* without any profiling. The results for *Swim95* differ in one of the four non-trivial groups.

Table 4 shows the performance difference between the two layouts on IBM and Intel machines. At “-O3”, the compiler analysis gives better improvement than distance-based profiling. The two layouts have the same performance at “-O5”, the highest optimization level. Without any profiling, the frequency-based affinity analysis is as effective as distance-based affinity analysis.

**Table 4: Comparison of frequency and K-distance analysis on *Swim95***

			Frequency	K-distance
Groups			unew, vnew, pnew u,v uold,vold,pold cu,cv,z,h	unew, vnew, pnew u,v uold,pold cu,cv,z,h
IBM	-03	time	17.1s	17.6s
		speedup	96%	90%
	-05	time	15.2s	15.3s
		speedup	91%	91%
Intel	-03	time	41.2s	42.3s
		speedup	48%	44%
	-05	time	34.9s	34.9s
		speedup	42%	42%

### Comparison with lightweight profiling

The lightweight profiling gives the execution frequency of loop bodies and call sites. These numbers are used to calculate data-access vectors. The resulting affinity groups are the same compared to the pure compiler analysis. Therefore, code profiling does not improve the regrouping results of the analysis. One exception, however, is when a program is transformed significantly by the compiler. The profiling results reflect the behavior of the optimized program, while our compiler analysis measures the behavior of the source program. Among all test programs, *Swim2K* and *Swim95* are the only ones in which binary-level profiling of the optimized program yields different affinity groups than compiler analysis.

The performance improvement from array regrouping Table 5 and Table 6 show the speedup on IBM and Intel machines, respectively. We include only programs where array regrouping is applied. Each program is compiled with both “-O3” and “-O5” optimization flags. At “-O5” on IBM machines, array regrouping obtained more than 10% improvement on *Swim2K*, *Swim95*, and *Tomcatv*, 2-3% on *Applu* and *Facerec*, and marginal improvement on *Galgel* and *Mgrid*. The improvement is significantly higher at “-O3”, at least 5% for all but *Mgrid*. The difference comes from the loop transformations, which makes array access more contiguous at “-O5” and reduces the benefit of array regrouping. The small improvements for *Facerec* and *Mgrid* are expected because only small arrays show reference affinity.

Our Intel machines did not have a good FORTRAN 90 compiler, so Table 6 shows results for only FORTRAN 77 programs. At “-O5”, array regrouping gives similar improvement for *Swim2K* and *Swim95*. It is a contrast to the different improvement on IBM, suggesting that the GNU compiler is not as highly tuned for SPEC CPU2000 programs as the IBM compiler is. *Applu* runs slower after array regrouping on the Intel machine. The regrouped version also runs 16% slower at “-O5” than “-O3”. We are investigating the reason for this anomaly.

**Table 5: Execution time (sec.) on IBM® POWER4™**

Benchmark	-03 Optimization		-05 Optimization	
	Original	Regrouped (speedup)	Original	Regrouped (speedup)
Applu	176.4	136.3 (29.4%)	161.2	157.9 (2.1%)
Facerec	148.6	141.3 (5.2%)	94.2	92.2 (2.2%)
Galgel	123.3	111.4 (10.7%)	83.2	82.6 (0.7%)
Mgrid	231.4	230.1 (0.6%)	103.9	103.0 (0.9%)
Swim2K	236.8	153.7 (54.1%)	125.2	110.1 (13.7%)
Swim95	33.6	17.1 (96.5%)	29.0	15.2 (90.8%)
Tomcatv	17.3	15.4 (12.3%)	16.8	15.1 (11.3%)

**Table 6: Execution time (sec.) on Intel® Pentium® 4**

Benchmark	-03 Optimization		-05 Optimization	
	Original	Regrouped (speedup)	Original	Regrouped (speedup)
Applu	427.4	444 (-3.7%)	429.4	444.6 (-3.4%)
Facerec	-	-	-	-
Galgel	-	-	-	-
Mgrid	461.7	460.6 (0.20%)	368.9	368.1 (0.2%)
Swim2K	545.1	315.7 (72.70%)	408.8	259.4 (57.6%)
Swim95	61.1	41.2 (48.30%)	49.7	34.9 (42.4%)
Tomcatv	48.8	44.5 (9.70%)	40.9	37.8 (8.2%)

### The choice of the affinity threshold

In the experiment, the affinity threshold is set at 0.95, meaning that for two arrays to be grouped, the normalized Manhattan distance between the access-frequency vectors is at most 0.05. To evaluate how sensitive the analysis is to this threshold, we apply X-means clustering to divide the affinity values into groups. Table 7 shows the lower boundary of the largest cluster and the upper boundary of the second largest cluster. All programs show a sizeable gap between the two clusters, 0.13 for *Apsi* and more than 0.2 for all other programs. Any threshold between 0.87 and 0.99 would yield the same affinity groups. Therefore, the analysis is quite insensitive to the choice of the threshold.

**Table 7: The affinities of the top two clusters**

Benchmark	Cluster-I lower boundary	Cluster-II upper boundary
Applu	0.998	0.667
Apsi	1	0.868
Facerec	0.997	0.8
Galgel	1	0.8
Lucas	1	0.667
Mgrid	1	0.667
Swim	0.995	0.799
Swim95	0.995	0.799
Tomcatv	1	0.798
Wupwise	1	0.8

## 5. RELATED WORK

In the context of parallelization, Kennedy and Kremer [9], Anderson et al. [2], and Jeremiassen and Eggers [8] used data transformation to improve locality for parallel programs. Cierniak and Li [4] combined loop and data transformations to improve locality, an approach that was later used by other researchers. The goal of most of these techniques is to improve data reuse within a single array. An exception is *group and transpose*, which groups single-dimension vectors used by a thread to reduce false sharing [8]. Grouping all local data may reduce cache spatial locality if they are not used at the same time.

Ding and Kennedy used array regrouping to improve program locality [5]. Their technique interleaves the elements of two arrays if they are always accessed together. They later developed partial regrouping for high-dimensional arrays [6]. These methods are conservative and do not consider the frequency of the data access and the control flow inside a loop nest. Their compiler could not handle most SPEC floating-point benchmarks. Recently, Zhong et al. defined the concept of reference affinity at the trace level and gave a profiling-based method for array regrouping and structure splitting [15]. The profiling requires measuring the reuse distance of each data access on a trace and attributes the result to the source-level data structures. For three FORTRAN programs, they showed that the profiling-based regrouping outperformed the compiler-based regrouping by an average of 5% on IBM® POWER4™ and 8% on Intel® Pentium® 4. However, the profiling analysis is several orders of magnitude slower than a normal program execution. In this paper, we have developed a new method that is more aggressive than the previous compiler technique and far less costly than the previous profiling method. In addition, we test the technique on most SPEC floating-point benchmarks.

Locality between multiple arrays can be improved by array padding [3,10], which changes the space between arrays or columns of arrays to reduce cache conflicts. In comparison, data regrouping is preferable because it works for all sizes of arrays on all configurations of cache, but padding is still needed if not all arrays can be grouped together.

## 6. CONCLUSIONS

Affinity analysis is an effective tool for data layout transformations. This work has developed a frequency-based affinity model, a context-sensitive interprocedural analysis, static estimate and lightweight profiling of the execution frequency. The analysis methods are implemented in a production compiler infrastructure and tested on SPEC CPU2000 benchmark programs. Array regrouping improves the performance for the majority of programs tested. The pure compiler analysis performs as well as data or code profiling does. These results show that array regrouping is an excellent candidate for inclusion in future optimizing compilers.

## 7. REFERENCES

- [1] R. Allen and K. Kennedy. Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers, October 2001.
- [2] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Santa Barbara, CA, July 1995.
- [3] D. Bailey. Unfavorable strides in cache memory systems. Technical Report RNR-92-015, NASA Ames Research Center, 1992.
- [4] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, California, June 1995.
- [5] C. Ding and K. Kennedy. Inter-array data regrouping. In Proceedings of The 12th International Workshop on Languages and Compilers for Parallel Computing, La Jolla, California, August 1999.
- [6] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. Journal of Parallel and Distributed Computing, 64(1), 2004.
- [7] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 1994.
- [8] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 179–188, Santa Barbara, CA, July 1995.
- [9] K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. ACM Transactions on Programming Languages and Systems, 20(4), 1998.
- [10] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 1998.
- [11] V. Sarkar. Determining average program execution times and their variance. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, Oregon, January 1989.
- [12] R. Silvera, R. Archambault, D. Fosbury, and B. Blainey. Branch and value profile feedback for whole program optimization. Unpublished, no date given.

- [13] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 1994.
- [14] M. Wegman and K. Zadeck. Constant propagation with conditional branches. In Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages, New Orleans, LA, January 1985.
- [15] 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.
- [16] K. McKinley, O. Temam. A quantitative analysis of loop nest locality. In Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, Cambridge, MA, US, 1996

### **Trademarks**

IBM and POWER4 are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.