Data-Layout Optimization Using Reuse Distance Distribution

Xiong Fu, Yu Zhang, and Yiyun Chen

Department of Computer Science and Technology,
University of Science and Technology of China
Hefei, Anhui 230027, China
fuxiong@mail.ustc.edu.cn, {yuzhang, yiyun}@ustc.edu.cn

Abstract. As the ever-increasing gap between the speed of processor and the speed of memory has become the cause of one of primary bottlenecks of computer systems, modern architecture systems use cache to solve this problem, whose utility heavily depends on program data locality. This paper introduces a platform independent data-layout optimization framework to improve program data locality. This framework uses a variable relation model based on variables' reuse distance distribution to quantitate the relation of variables and accordingly assigns variables which are often accessed together in one group. At the same time this framework introduces a dynamic array regrouping method to group dynamic arrays assigned in a group. Experiments show that this data-layout optimization framework can effectively improve program data locality and program performance.

1. Introduction

Over the past 30 years, computer processor speed has been increasing about 60% annualy, but the increasing rate of memory speed has been less than 10% per year. The ever-increasing gap between the speed of processor and the speed of memory has become the cause of one of the primary bottlenecks of computer system [1]. Now many computer programs and systems widely employ cache to decrease the impact of the gap and cache performance has an increasing influence on system speed, cost and energy usage [9]. The utility of cache mostly depends on program instruction locality and program data locality, especially the program data locality. Thus, optimizing program data locality and improving computer system performance have been an active area of research [2,3,4,5,6,7].

Past existing works provide mainly two ways to optimize program data locality: one is code transformation [2], such as loop tiling, loop fusion and loop interchange, which can significantly increase both temporal and spatial locality, however most of the code is too complex to be transformed; the other is data reorganization [5,6], such as structure field reordering, structure splitting and array regrouping, which reorganize the layout of data structures to improve spatial locality.

Data-layout optimization needs to know program cache behavior. While current static program analysis techniques are inadequate because of the complex control

flow and indirect data access [6], most existing works about data-layout optimization are either profile-guided [3,5,6] or exploiting programmer-supplied application knowledge [7]. In recent years, reuse distance has become a metric for program cache behavior [8], and reuse distance can be used to analyze and predict cache behavior [9], which shows reuse distance maybe a suitable model for profile-guided data-layout optimization.

In this paper, we propose a data-layout optimization framework for C language based on reuse distance distribution. Our framework takes use of source level transformation and can be employed on many platforms, including some embedded systems. The framework defines a variable relation model based on variables' reuse distance distribution to quantitate the relation of variables. Variables often accessed together sequentially will be found by this model and be assigned in one group, which later can be used by data reorganization. At the same time our framework builds an array regrouping method for dynamic array. Experiments show that this data-layout optimization framework can effectively improve program data locality and program performance.

The rest of the paper is organized as follows. Section 2 introduces reuse distance. Section 3 gives an overview of the data-layout optimization framework. Section 4 presents main processes about variable analysis. Section 5 describes the compiler support for dynamic array regrouping. The last three sections present the experimental evaluation, related work, and conclusions, respectively.

2. Reuse Distance

In 1970, Mattson et al. introduced the concept of stack distances to analyze the behavior of demand paged memory systems and evaluate the performance of memory management schemes. To describe program cache behavior using architecture independent characteristic, Ding uses reuse distance substitute for stack distance [9].

Definition 1: data element is the sign of data accessed by program at run time, which may be memory address or memory region. We use letters such as *a, b, c* to represent data elements.

Definition 2: In a sequential execution, reuse distance is the number of distinct data elements accessed between two consecutive references to the same data element. It measures the volume of the intervening data between two references.

For example, given a sequence of data references, *abcdeaedb*, the reuse distance is $\infty\infty\infty\infty\infty 4124$ by the definition, while ∞ means the first time reference and there is no reuse. Program reuse distance is composed of reuse distance of all references of the whole program, which is normally expressed by histogram and is used to evaluate the data cache behavior and data locality of the whole program. A variable's reuse distance infers the reuse distance of references which access this variable's memory region, which can show some characteristics of variable reference.

If the data element of reuse distance is cache line, reuse distance indicates the number of distinct cache line accessed between two consecutive references to the same cache line. In a fully associative LRU cache with n lines, a reference with reuse distance $d < n$ will hit, while a reference with reuse distance $d \geq n$ will miss, by

which reuse distance can be used to analyze program cache miss rate [8]. Reuse distance can be employed not only to accurately analyze cache miss rate of fully associative LRU cache, but also to approximately analyze cache miss rate of set associative LRU cache.

3. Optimization Framework

Our data-layout optimization framework is depicted in Figure1, which includes two main parts: the variable analysis part and the data reorganization part. The variable analysis part is mainly composed of the following processes:

- **Source-Level Instrumentation**: source-to-source transformation for C source code, adding new code into source code to collect memory reference information at run time.
- **Reuse Distance Analysis**: analyzing the reuse distance of memory reference.
- **Variable Relation Analysis**: analyzing variables' reuse distance distribution, using variable relation model to quantitate relation of variables, and assigning variables which are often accessed together in one group

The data reorganization part is composed of a source-to-source transformer based on the information of variable analysis. In the transformation the data layout is reorganized to improve data locality. Now our transformer only supports dynamic array regrouping.

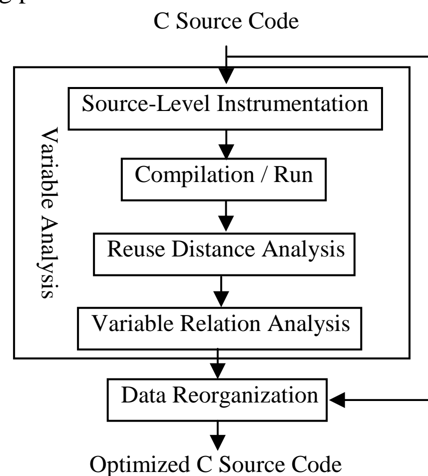


Fig.1. Optimization framework

4. Variable Analysis

4.1 Source-Level Instrumentation

Our framework takes use of source-level instrumentation to collect program memory reference information, including address and variable information. We finish a source-to-source transformer based on the SUIF compiler infrastructure of Stanford University [10] to do instrumentation. When source codes are transformed, new code is added to record memory access information. The addresses of variables, including the pointer dereference, array, scalar variables, are profiled as memory access address. There are some differences with actual memory reference information at run time: (1) because of the register file, reference to some scalar variables doesn't access memory; (2) there are many times of memory access in a variable reference, such as copy operation of structure. Reuse distance of most scalar variables is small and the instance of the second difference is little in normal program, at the same time we only

care the relative difference of reuse distance distribution, so all difference can be ignored.

4.2 Reuse Distance Analysis

When computing reuse distance, we have found two important characteristics: (1) reuse distance of most references is of small value; (2) only a few of data elements in stack bottom (we use a stack to compute reuse distance) can be reused. If we limit the search length in reuse distance analysis, we only need a limited memory size to save the reference record and the search time will be limited also. We call this kind of reuse distance as limited reuse distance.

In actual reuse distance computation, we use probable max cache size as the search limit length. Here gives the algorithm of limited reuse distance (using a stack to compute reuse distance):

Repeat following steps for each memory reference $\chi_\tau\ (0 \le \tau < N)$:

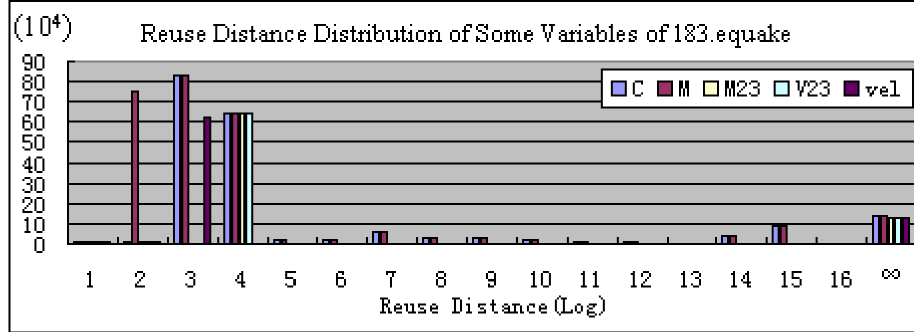
1) **Search**: find the location in the stack of the most recent reference to the address accessed by current reference χ_τ. The finding distance is no more than the max cache size;
2) **Count**: compute the reuse distance $dist(\tau)$ of current reference χ_τ. If the most recent reference exists in (1), count the number of elements on the stack above it as the value of $dist(\tau)$; otherwise, $dist(\tau)$ is defined as ∞;
3) **Update**: bring current reference χ_τ to the top of the stack. If the most recent reference exists in (1), delete it; if the length of stack is more than max cache size, remove the reference at the bottom of the stack.

Limited reuse distance limits the stack size to no more than max cache size, which has two advantages: for one thing, the space of algorithm is limited and doesn't increase with the reference number increase; for another, the time of algorithm is linear to the reference number, and the speed of reuse distance analysis is increased.

4.3 Variable Relation Analysis

Reuse distance distribution of variables always accessed together is very similar [6], which can be used to group program variables and improve program data locality. In this section we introduce how to group variables by their reuse distance in our framework, which can be divided into the following steps:

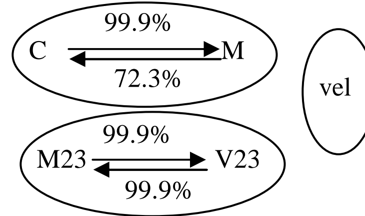
The first step is variable's reuse distance grouping. We study the distribution regulation of reuse distance by grouping. Figure 2(a) uses a histogram to express our reuse distance grouping example of five variables of 183.equake: the x-axis is a sequence of bins representing difference ranges of reuse distance (the value of x-axis infers the region $[2^{i-1}, 2^i)$); the y-axis gives the number of memory references whose reuse distance falls into each region (0 means reuse to itself, so we don't show it; and memory reference whose reuse distance is ∞ means over max cache size). We can see that reuse distance distribution of M23 and V23 is very similar, and reuse distance distribution of M and C is very similar except one bin.



(a) Variables' reuse distance distribution

	C	M	M23	V23	vel
C	100%	99.9%	40.1%	40.1%	39.0%
M	72.3%	100%	29.0%	29.0%	28.2%
M23	100%	99.9%	100%	99.9%	17.8%
V23	100%	99.9%	99.9%	100%	17.8%
vel	99.9%	99.9%	18.3%	18.3%	100%

(b) R value table of variables



(c) Groups of variables

Fig.2 Variable analysis based on reuse distance distribution

The second step is computing R value of variable. R value of variable is defined by variable relation model and is employed to quantitate relation of variables. Definition of variable relation model is as follows: The relation between variable i and variable j can be expressed by R(i, j), and R(i, j) can be computed by following formulas:

$$S_k(i, j) = Min(N_k(i), N_k(j)) \quad (1)$$

$$R(i, j) = \sum_{k=1}^{G} S_k(i, j) \Big/ \sum_{k=1}^{G} N_k(i) \quad (2)$$

In formula (1), $N_k(i)$ infers memory reference number of the *kth* group of variable i. $S_k(i, j)$ is min value of memory reference number of the *kth* group between variable i and variable j, which indicates the same part of the *kth* bin. In formula (2), G means number of groups, which also equals the number of bin. R(i, j) implies the ratio that the same part of i's reuse distance distribution and j's reuse distance distribution takes in reuse distance distribution of variable i. Figure 2(b) gives the R value table of five variables of 183.equake, and figure 3 shows the algorithm of computing R value.

The third step is variable grouping. R(i, j) and R(j, i) reflect relation between variable i and variable j, so we use value of R(i, j) + R(j, i) to describe relation between variable i and variable j. if the value is bigger, so will be the chance of variable i and variable j be accessed together, by which we can search the variables which are often sequentially accessed. Figure 2(c) shows result of variable grouping of five variables of 183.equake. Figure 3 gives the algorithms of variable regrouping.

```
Computing_R_Value(RDi, RDj){
  //RDi is reuse distance distribution array of i
  //RDj is reuse distance distribution array of j
  //N is length of arrary RDi
  //R is R value table

    Refi = 0;
    Refj = 0;
    Refij = 0;

    for (int k=0; k<N; k++){
       if (RDi[k]>RDj[k]){
          Refij += RDj[k];
       }else{
          Refij += RDi[k];
       }//end of if
       Refi += RDi[k];
       Refj += RDj[k];
    }//end of for

    R(i, j) = Refij / Refi;
    R(j, i) = Refij / Refj;
}
```

```
Variable_Grouping(G, max){
   //G is variable name set
   //max is limit value of variable relation
   //R is R value table
   while (G≠ ∅ ){
      choose s∈ G;
      T = {s};
      G = G\{s};//delete s from G
      exist = true;
      while (exist){
         exist = false;
         for each g∈ G{
            for each t∈ T{
               if (R(g, t) +R(t, g) >= max){
                  T = T ∪ {g};
                  G = G\{g};
                  exist = true;  break;
               }//end of if
            }//end of for
            if (exist) break;
         }//end of for
      }//end of while
      save group T;
   }//end of while
}
```

Fig.3 Computing R value and variable grouping

5. Data Reorganization

By the variable analysis, we can find variables which are often accessed together. In C programs, the attributes of a structure are stored together. If we reorganize the variables given by analysis result of section 4, program data locality can be improved. There are some existing data reorganization technologies for different data structures, such as array regrouping for array, structure splitting and structure field reordering for structure [6]. In this section we will introduce a regrouping technology for dynamic array.

Our dynamic array regrouping technology includes two main steps: the first is memory allocation merging, by allocating a uniform memory region for an array and substituting pointer for array access; the second is normal array regrouping [6]. We implement a source-to-source transformer based on the SUIF compiler infrastructure of Stanford University [10] to do dynamic array regrouping. The transformer changes the allocation and the reference of the dynamic arrays. Figure 4 gives an example of two-dimensional dynamic array regrouping. Initially M23, C23 and V23 are three two-dimensional dynamic arrays whose elements are double type. After dynamic array regrouping they can be grouped into a region indicated by a pointer.

In dynamic array regrouping there are two problems: First, the dimensions and the each dimension's size of dynamic array must be the same, but the type of element of array can not be the same. All of these are profiled by source-level instrumentation in

our framework. Second, in program transformation, the alias of dynamic array must be considered. In our framework points-to analysis in [11] is implemented to solve this problem.

```
//definition and initialization
    double **M23, **C23， **V23；
    M23 = (double **) malloc(ARCHnodes * sizeof(double *));
    C23 = (double **) malloc(ARCHnodes * sizeof(double *));
    V23 = (double **) malloc(ARCHnodes * sizeof(double *));
    for (i = 0; i < ARCHnodes; i++) {
        M23[i] = (double *) malloc(3 * sizeof(double));
        C23[i] = (double *) malloc(3 * sizeof(double));
        V23[i] = (double *) malloc(3 * sizeof(double));
    }
//reference
disp[disptplus][i][j] += 2.0*M[i][j]*disp[dispt][i][j]-(M[i][j]-Exc.dt/2.0*C[i][j]) *
                  disp[disptminus][i][j] -Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0+
                                    C23[i][j] * phi1(time) / 2.0 +V23[i][j] * phi0(time) / 2.0);
```

(a) Definition and reference before regrouping

```
//definition and initialization
    struct MCV{
        double M23;
        double C23;
        double V23;
    };
struct MCV *pmcv = (struct MCV*)malloc(ARCHnodes*3*sizeof(struct MCV));
//reference
disp[disptplus][i][j] += 2.0 * M[i][j] * disp[dispt][i][j]-(M[i][j]-Exc.dt/2.0* C[i][j])*
        disp[disptminus][i][j]-Exc.dt*Exc.dt*((*(pmcv+i*3+j)).M23*phi2(time)/2.0+
        (*(pmcv+i*3+j)).C23*phi1(time)/2.0+(*(pmcv+i*3+j)).V23*phi0(time)/2.0);
```

(b) Definition and reference after regrouping

Fig.4 Regrouping example of two-dimensional dynamic array (183.equake)

6. Experiments

As described earlier, we take use of two transformers based on SUIF [10] to do the source-level instrumentation and dynamic array regrouping. The transformed C code is compiled by GCC 3.2.2 at –O3. Experiments run on an Intel Celeron(R) (2.0G) processor running Red Hat Linux 9.0, which has 8K L1 data cache (4-way) and 128k L2 cache (2-way), and the cache line size is 64byte. Our testing programs are three programs from SPEC CPU2000 with some dynamic arrays: 183.equake, 179.art and 188.ammp. Our optimization framework analyzes relationship of all dynamic arrays at the *test* input, and regroups some dynamic arrays.

Table1 shows our optimization result. One program, 183.equake, has two groups including five dynamic arrays, and the other two programs, 179.art and 188.ammp, have only one group. For three programs with ten inputs, our framework gets speedup from 0.60% to 7.41%, and the average speedup is 4.22%.We find arrays grouped in

the first two programs are two dimensions, but arrays grouped in 188.ammp are one dimension. Therefore, the dimension of array is quite sensitive to optimization.

Table 1. Optimization Result

Benchmark	Grouped Array	Input	Memory Reference Times		Speedup
			Standard	Optimized	
183.equake	(M, C)	Test	542,890,080	539,682,079	**5.86%**
	(M23,C23,	train			**7.41%**
	V23)	ref			**6.12%**
179.art	(tds, bus)	test	1,390,572,801	1,388,948,481	**6.74%**
		train			**7.36%**
		ref1			**2.74%**
		ref2			**3.49%**
188.ammp	(x, y, z,	test	2,733,856,332	2,733,633,990	**0.77%**
	xx, yy,	train			**1.10%**
	zz)	ref			**0.60%**
				Average	**4.22%**

To evaluate the impact on program data locality, we measure the reuse distance of three programs at *test* input, and analyze the cache miss based on reuse distance. In our experiments, Pin [12] is availed to trace program memory references. Memory Reference Times in table 1 shows the number of memory reference computed by Pin [12]. We can see that optimization reduces the number of memory reference.

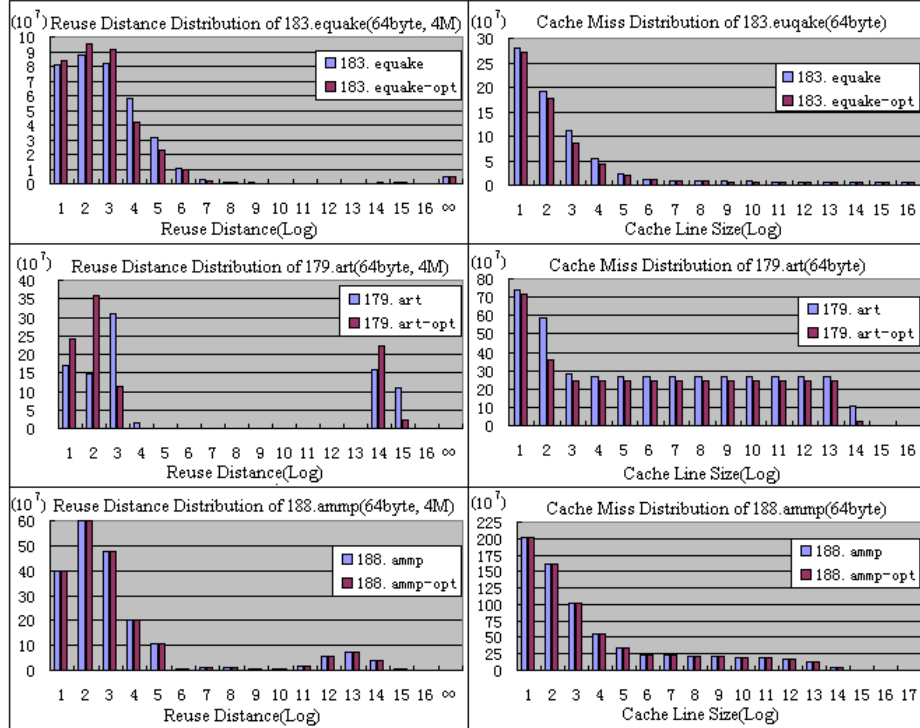


Fig. 5 Reuse distance and cache miss distribution at *test* input

Figure 5 gives the comparison of reuse distance and cache miss between normal program and optimized program. Both reuse distance and cache miss are given in *cache blocks*; multiplying by 64 converts the range to bytes. For reuse distance distribution, the x-axis shows the distance in a log scale (the max cache size is 4M), and the y-axis shows the number of memory reference. For cache miss distribution, the x-axis shows the cache size in a log scale, and the y-axis shows the number of cache miss. From figure 5, we can observe that the reuse distance of whole program decreases and the cache miss number reduces in all kinds of cache size after optimization.

7. Related Work

Program cache behavior analysis and data structure reorganization have been the subjects of much research. For example, T.M. Chilimbi [3] defines a cache behavior model named hot-streams, which uses the frequency of data sub-streams to quantitate relation of structure fields, and uses structure splitting to improve data locality. The model combines dynamic relation with frequency but does not give whole-program relation. Shai Rubin [5] proposes a parameterizable framework for data-layout optimization. Their framework finds out a good layout by searching the space of possible layouts, with the help of profile feedback, and takes use of field reordering and custom memory allocation to improve data locality.

Yutao Zhong [6] defines a cache behavior model called reference affinity based on reuse distance signature, which measures how close a group of data are always accessed together in a reference trace. When applied for array regrouping and structure splitting, their model can effectively improve program data locality and program performance. This paper demonstrates that a similar model can serve as a metric of data structure reorganization. Together, we provide a new array regrouping method for dynamic array.

8. Conclusion and Future Work

In this paper we present a data-layout optimization framework. Unlike prior works on data-layout optimization, our framework uses a variable relation model based on variables' reuse distance distribution to find variables which are often accessed together. In addition, our framework introduces a new data reorganization technology for dynamic array to improve data locality. Our framework takes use of source code transformation and is platform independent. Experiments show that this framework can optimize program data locality and improve program performance. Three test programs have gotten an average speedup of 4.22% by improving data locality.

Our future work includes: (1)optimizing the variable relation model, and enabling it to find variable relation more accurately; (2)implementing more data reorganization on our data-layout optimization framework, such as normal array regrouping, structure splitting, and structure field reordering.

Acknowledge

The authors would like to thank Xiaofeng Li, Yan Guo for their useful discussions and the anonymous referees for their useful comments. This work is supported by the National Natural Science Foundation of China under Grant No. 60473068, with support from Intel China Research Center.

Reference

[1] David Patterson, Thomas Anderson, Neal Cardwell et al. A Case for Intelligent RAM. In IEEE Micro, Apr 1997, pp 34–44.

[2] K. S. McKinley et al. Improving Data Locality with Loop Transformations. ACM Transactions on Programming Languages and Systems, 18(4):424-453.

[3] T.M. Chilimbi. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In Proceedings of PLDI’01, pp. 191-202.

[4] Youfeng Wu. Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching. In Proceedings of PLDI’02, June 2002, pp210-221.

[5] Shai Rubin, Rastislav Bodik, Trishul Chilimbi. An Efficient Profile-Analysis Framework for Data-Layout Optimizations. In Proceedings of POPL’02, 2002, pp 140-153

[6] Yutao Zhong, Maksim Orlovich, Xipeng Shen, Chen Ding. Array Regrouping and Structure Splitting using Whole-Program Reference Affinity. In Proceedings of PLDI’04, June 2004, pp. 255 - 266.

[7] K. Beyls et al. Platform-Independent Cache Optimization by Pinpointing Low-Locality Reuse. In Proceedings of ICCS’04, volume 3, pp 463–470.

[8] K. Beyls, E. D'Hollander. Reuse Distance as a Metric for Cache Behavior. In Proceedings of the Conference on Parallel and Distributed Computing and Systems, 2001, pp. 617–662

[9] Chen Ding, Yutao Zhong. Predicting Whole-Program Locality with Reuse Distance Analysis. In Proceedings of PLDI’03, San Diego, CA, June 2003.pp.245-257.

[10] B.P. Wilson et al. SUIF: A Parallelizing and Optimizing Research Compiler. ACM SIGPLAN Notices, 29(12):31-37, December 1994.

[11] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. Proceedings of POPL’96, St. Petersburg, FL, Jan 1996.

[12] Chi-Keung Luk, Robert Cohn, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In Proceedings of PLDI’05, Chicago, Illinois, USA, June 2005, pp. 190-200