A Framework for Safe Automatic Data Reorganization

Shimin Cui (Speaker), Yaoqing Gao, Roch Archambault, Raul Silvera
IBM Toronto Software Lab

Peng Peers Zhao, Jose Nelson Amaral
University of Alberta

* Part of the work joint with Xipeng Shen and Chen Ding (University of Rochester)

Contents

- The memory wall problem
- Data reorganization framework
  - Data reshape analysis
  - Data reshape planning
- Performance evaluation
- Summary

The Memory Wall Problem

- Memory access latency is a “wall” to better performance
  - Speed of memory continues to lag behind the speed of processors.
- Memory hierarchy
  - Limited cache size and bandwidth limitation
  - Efficient utilization of cache is crucial for performance
- Domain – applications and benchmarks with large data sets.

Compiler Approaches

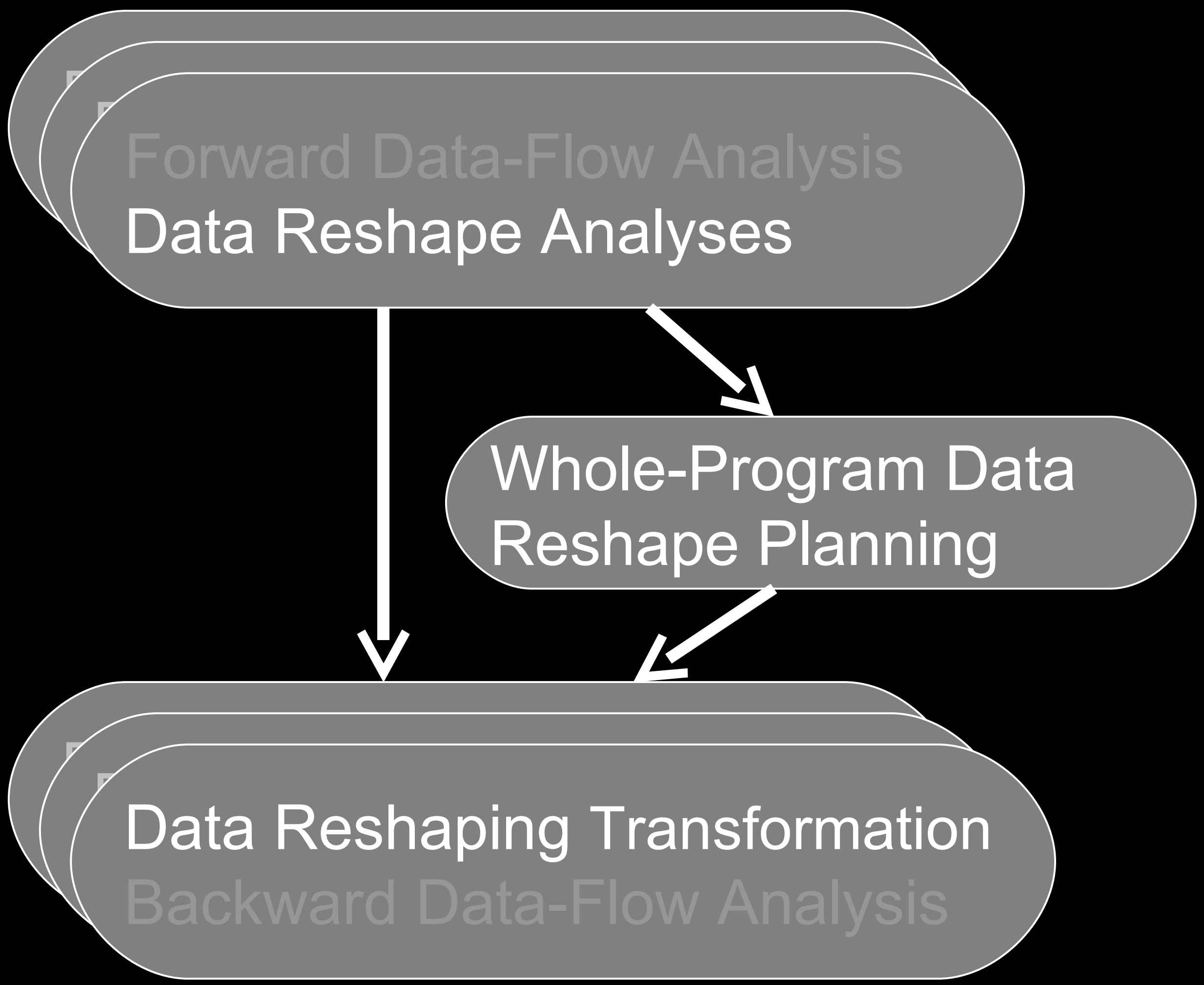
- Tolerate memory latency through buffering and pipelining data references.
  - Data Prefetching
- Reduce memory latency through locality optimizations.
  - Code transformations - modifying the actual algorithm by reordering computations.
    - ✓ Loop fusion
    - ✓ Loop distribution
    - ✓ Loop tiling/blocking
    - ✓ Loop interchange
  - Data reorganizations - Placing data in memory according to their access patterns.

Data Reorganization

- Data reorganization is a difficult problem.
  - NP problem, heuristics are needed
  - Safe, automatic techniques are necessary
- Data layout transformations
  - Data splitting
  - Fields reordering
  - Data interleaving
  - Data padding
  - Data coalescing
  - …

Data Reorganization Framework

- Inside TPO link time optimizations



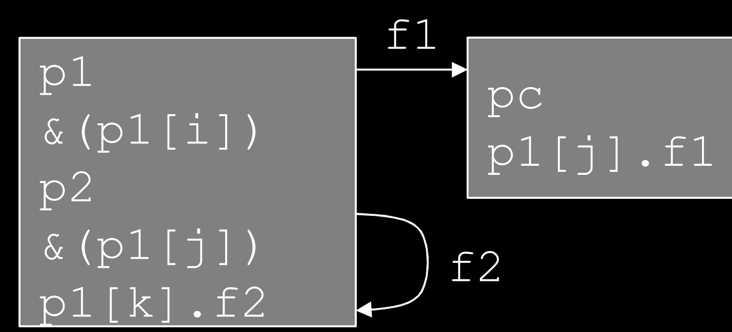
Data Reshape Analyses

- Safety and legality issues
  - Inter-procedural alias analysis
    - Pointer escape analysis
    - Global pointer analysis
  - Data shape analysis
- Profitability issues
  - Data affinity analysis

Inter-procedural Alias Analysis

- To ensure data reshaping correctness since compiler needs to modify all the affected references when it reshapes a data object and its aliases.
- Flow-insensitive alias analysis is sufficient for data reshaping [Steensgaard].
- Field sensitive alias analysis is necessary to trace and distinguish the alias relationships among different fields.

```
char *pc;
struct A {
  char *f1;
  struct A *f2;
} *p1, *p2;
p1 = malloc(N * sizeof(A));
p2 = &(p1[i]);
p1[j].f1 = pc;
p1[k].f2 = &(p1[i]);
```



Storage shape graph

Data Shape Analysis

- Reshaping on data that has incompatible type is unsafe and is strictly avoided.
- Type compatibility analysis is integrated with the interprocedural alias analysis.
  - The interprocedural alias analysis keeps track of the type of each alias set.
  - The types of data in an alias set must be compatible in the whole program for safe data reshaping.
- Compatibility rules are enforced to check the access patterns. Two data types are compatible if
  - Two intrinsic data types are compatible if their data lengths are identical.
  - Two aggregated data structures are compatible if they have the same number of byte-level fields and their corresponding fields have the same offset and length.
  - Two arrays have compatible types if their element types are compatible, they have the same dimensions and the strides of corresponding dimensions are also identical.
  - Two pointers are of compatible types iff the data they point to have compatible types.

Lightweight Data Affinity Analysis
(Joint Work With Xipeng Shen and Chen Ding)

- To measure how closely a group of data are accessed together in a program region.
- Model affinity based on access frequency:
  - An access frequency vector AFV(A) is used for each data to record all the access frequency in all the innermost loops in the program.
  - Unique data is identified based on alias analysis, and AFVs of their aliases are merged.
  - Two data have good affinity if their AFVs are similar:

$$affinity(A,B) = 1 - \sum_{i=1}^{N} |(f_i(A) - f_i(B)| / (0.0001 + \sum_{i=1}^{N} (f_i(A) + f_i(B)))$$

N - # of innermost loops, $f_i(A)$ – access frequency of A in i-th loop

  - Construct and partition data affinity graph to obtain all the affinity groups.

Data Reshaping Planning

- Based on the reshape analysis and affinity analysis, a plan is made how to reshape a data.
  - Array splitting
  - Data outlining
  - Data allocation merging
  - Data interleaving
  - …

Array Splitting

- Separate cold fields from hot fields to avoid bringing rarely accessed data into the cache in order to increase cache utilization.
  - A structure array is split into several contiguous arrays.
  - Fields are reordered based on affinity information for large data structure.
- Target to aggregate arrays that have consistent compatible access patterns.
- Three approaches:
  - Affinity-based splitting
  - Frequency-based splitting
  - Maximal data splitting

Array Splitting – Three Approaches

- Original data structure
  - hot (F0,F2, F3), affinity groups (F0, F3) (F2), (F1, F4)

F0	F1	F2	F3	F4

- Original array [4]

$F0_0$	$F1_0$	$F2_0$	$F3_0$	$F4_0$	$F0_1$	$F1_1$	$F2_1$	$F3_1$	$F4_1$	$F0_2$	$F1_2$	$F2_2$	$F3_2$	$F4_2$	$F0_3$	$F1_3$	$F2_3$	$F3_3$	$F4_3$

- Affinity-based splitting

$F0_0$	$F3_0$	$F0_1$	$F3_1$	$F0_2$	$F3_2$	$F0_3$	$F3_3$	$F2_0$	$F2_1$	$F2_2$	$F2_3$	$F1_0$	$F4_0$	$F1_1$	$F4_1$	$F1_2$	$F4_2$	$F1_3$	$F4_3$

- Frequency-based splitting

$F0_0$	$F3_0$	$F2_0$	$F0_1$	$F3_1$	$F2_1$	$F0_2$	$F3_2$	$F2_2$	$F0_3$	$F3_3$	$F2_3$	$F1_0$	$F4_0$	$F1_1$	$F4_1$	$F1_2$	$F4_2$	$F1_3$	$F4_3$

- Maximal data splitting

$F0_0$	$F0_1$	$F0_2$	$F0_3$	$F3_0$	$F3_1$	$F3_2$	$F3_3$	$F2_0$	$F2_1$	$F2_2$	$F2_3$	$F1_0$	$F1_1$	$F1_2$	$F1_3$	$F4_0$	$F4_1$	$F4_2$	$F4_3$

Array Splitting - Static Arrays

```
struct {
  double x;
  double y;
} a[1000];

void foo() {
  for (i=0; i<N; i++) {
    … = a[i].x …;
  }
  …
  for (i=0; i<M; i++) {
    … = a[i].y …;
  }
}
```

```
double ax[1000];
double ay[1000];


void foo() {
  for (i=0; i<N; i++) {
    … = ax[i] …;
  }
  …
  for (i=0; i<M; i++) {
    … = ay[i] …;
  }
}
```

Array Splitting – Single-Instantiated Dynamic Arrays

```
typedef struct {
  double x;
  double y;
} S;
S *p;


void init () {
  p = malloc(sizeof(S)*N);
}



void foo() {
  for (i=0; i<N; i++) {
    … = p[i].x …;
  }
  …
  for (i=0; i<N; i++) {
    … = p[i].y …;
  }
}
```

```
typedef struct {
  double x;
  double y;
} S;
void *p;
double *xbasep, *ybasep;

void init () {
  p = malloc(sizeof(S)*N);
  xbasep = p;
  ybasep = xbasep + sizeof(double)*N;
}

void foo() {
  for (i=0; i<N; i++) {
    … = xbasep[i] …;
  }
  …
  for (i=0; i<N; i++) {
    … = ybasep[i] …;
  }
}
```

Array Splitting – Multiple-Instantiated Dynamic Arrays

- Runtime descriptor is introduced to handle the multiple instantiations.

```
typedef struct {
  double x;
  double y;
} S;
S *p, *q;

void init (N) {
  p = malloc(sizeof(S)*N);
}

void bar() {
  init(N1);
  q = p;
  init(N2);
}

void foo() {
  for (i=0; i<N2; i++) {
    … = p[i].x …;
  }
  …
  for (i=0; i<N1; i++) {
    … = q[i].y …;
  }
}
```

→

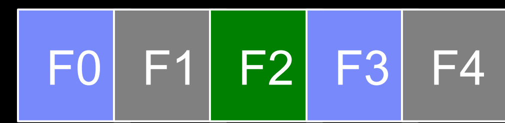
```
typedef struct {              typedef struct {
  double x;                     void *basep;
  double y;                     double *xbasep;
} S;                            double *ybasep;
                              } desc;
                              desc *p, *q;

void init (N) {
  p = malloc(sizeof(desc) + sizeof(S)*N);
  p->basep = p + sizeof(desc);
  p->xbasep = p->basep;
  p->ybasep = p->xbasep + sizeof(double)*N;
}

void bar() { init(N1); q = p; init(N2);}

void foo() {
  for (i=0; i<N2; i++) {
    … = p->xbasep[i] …;
  }
  …
  for (i=0; i<N1; i++) {
    … = q->ybasep[i] …;
 }
}
```

Data Outlining

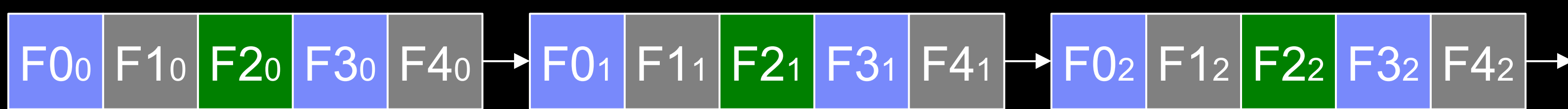
- Separate cold fields from hot fields to avoid bringing rarely accessed data into the cache in order to increase cache utilization.
- Target to non-array data objects whose collection of hot fields are smaller than the cache block size.
- The outlined fields must be cold.
- No need to worry about single/multiple object instantiations.

Data Outlining Approach

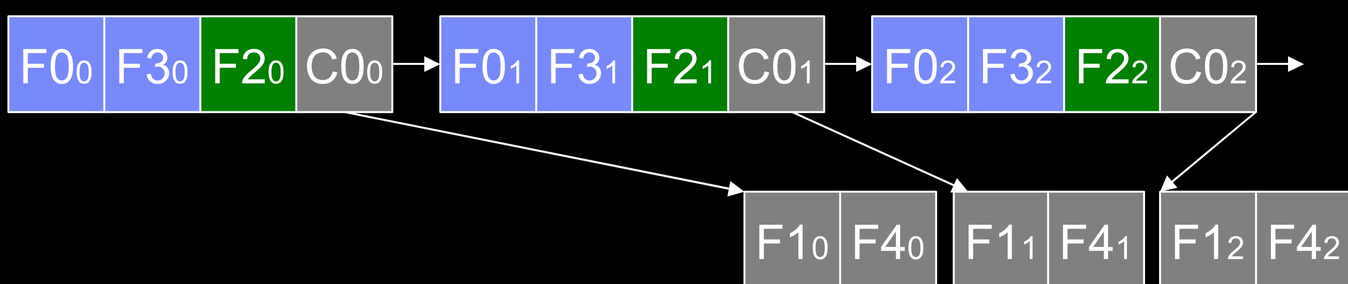
- Original linked list element:



- Original linked list



- Frequency-based outlining

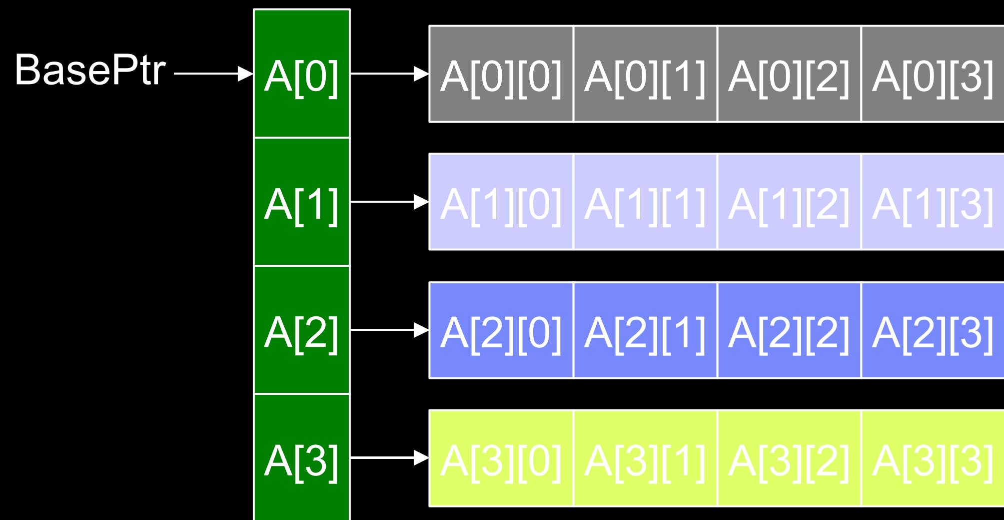


Data Allocation Merging

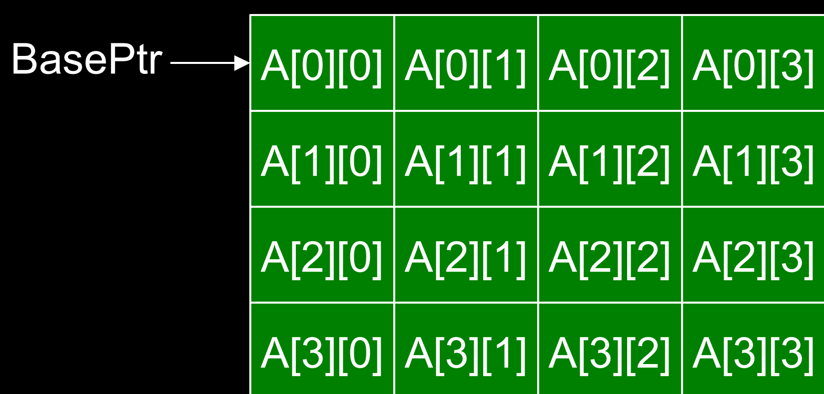
- Flat multi-dimensional dynamic array into contiguous memory space to achieve better reference locality.
- Target to multi-dimensional dynamic arrays with (almost) rectangular shapes. Padding is needed for non-rectangular shaped multi-dimensional dynamic arrays.
- Facilitate loop locality transformation since indirect reference is replaced by array indexed reference.
- Runtime descriptor is also introduced to handle the multiple object instantiation cases.

Data Allocation Merging Approach

- Original two dimensional dynamic array **A



- After data allocation merging *A’

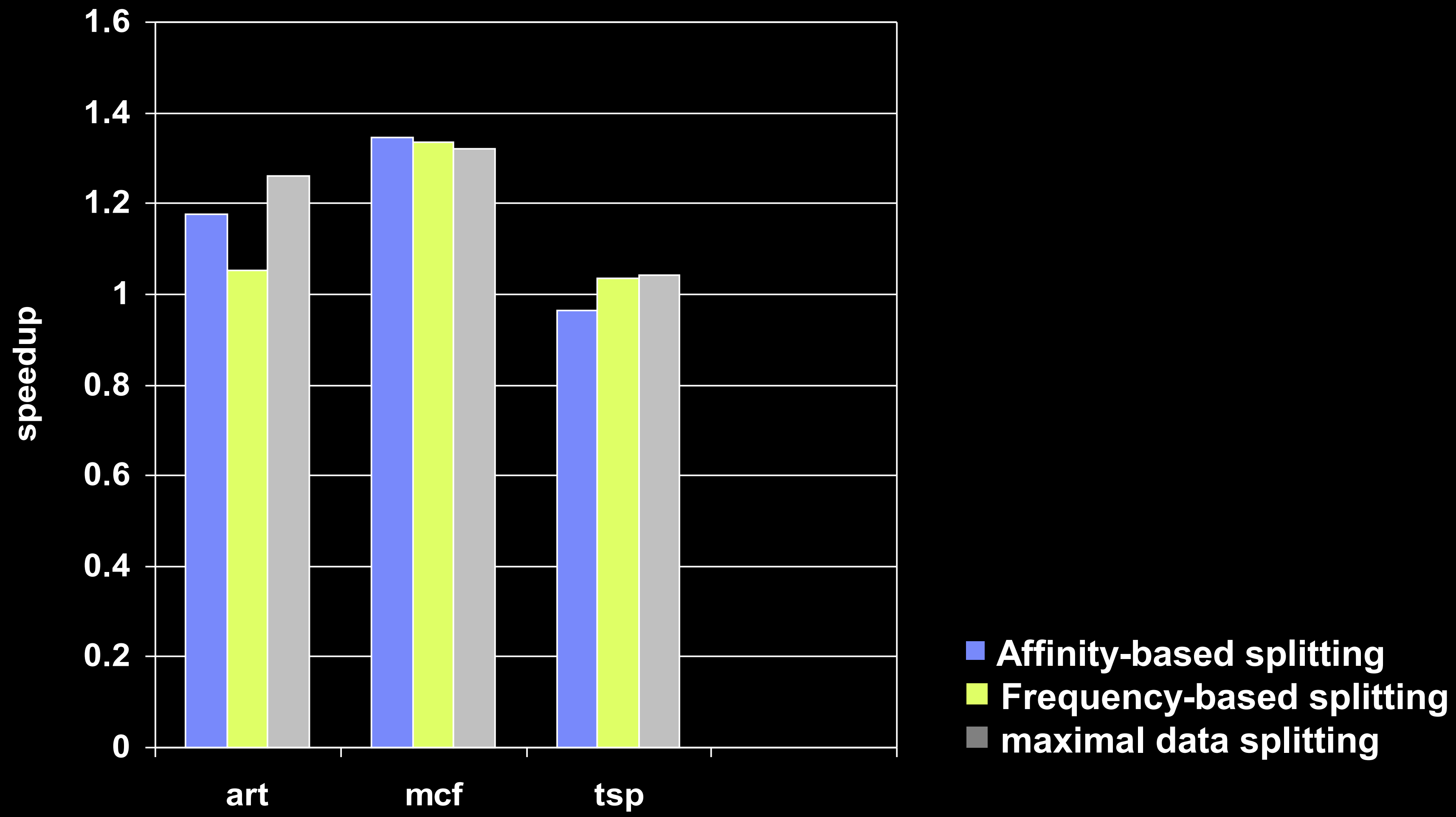


Data Allocation Merging – Dynamic Arrays

```
float **A = (float **) malloc(N*sizeof(float *));
float **B = (float **) malloc(N*sizeof(float *));
float *C[N];

for (i = 0; i < N; i++) {
    A[i] = (float *) malloc(N*sizeof(float));

    B[i] = (float *) malloc(N*sizeof(float));

    C[i] = (float *) malloc(N*sizeof(float));
}

for (j = 0; j < n; j++)
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      C[i][j] += A[i][k] * B[k][j];
    // *(C[i]+j) += *(*(A+i)+k) * *(*(B+k)+j)
```

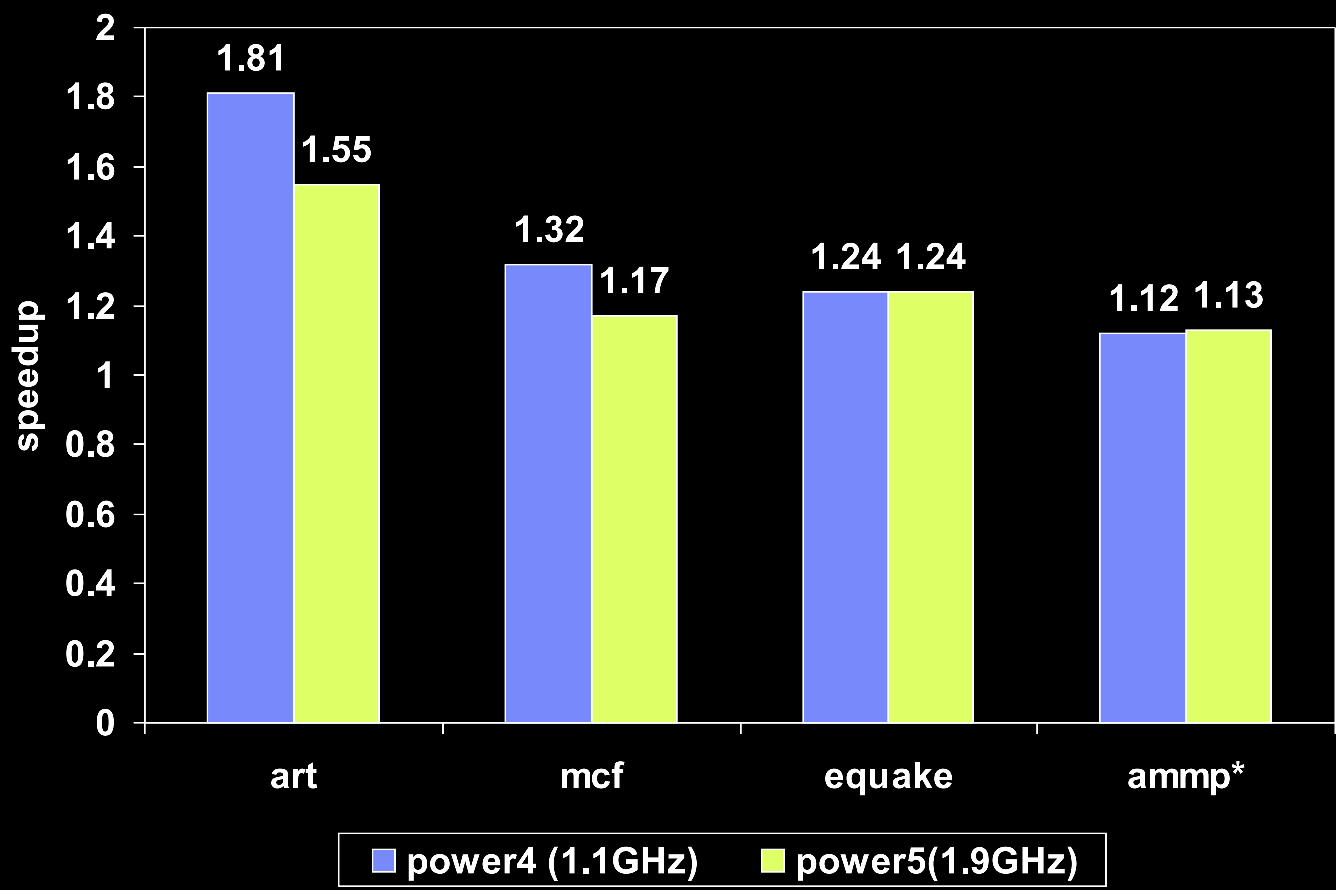
```
float *A = (float *) malloc(N*N*sizeof(float));
float *B = (float *) malloc(N*N*sizeof(float));
float *C = (float *) malloc(N*N*sizeof(float));

for (j = 0; j < N; j++)
  for (k = 0; k < N; k++)
    for (i = 0; i < N; i++)
      C[i][j] += A[i][k] * B[k][j];
    // *(C+i*N+j) += *(A+j*N+k) * *(B+k*N+j)
```

Comparison of Splitting Approaches

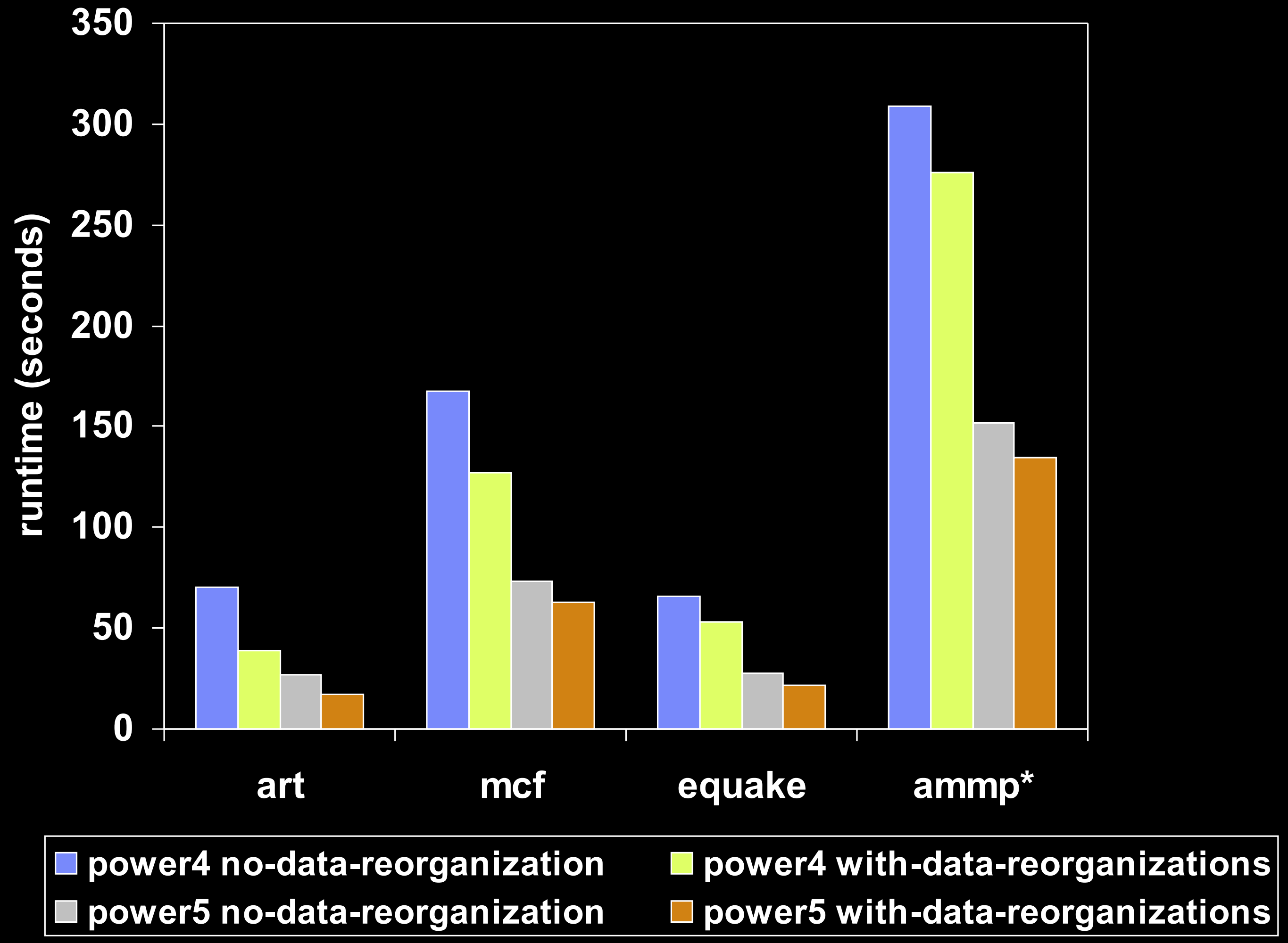


(Measured on power4 1.1GHz, AIX5.2)

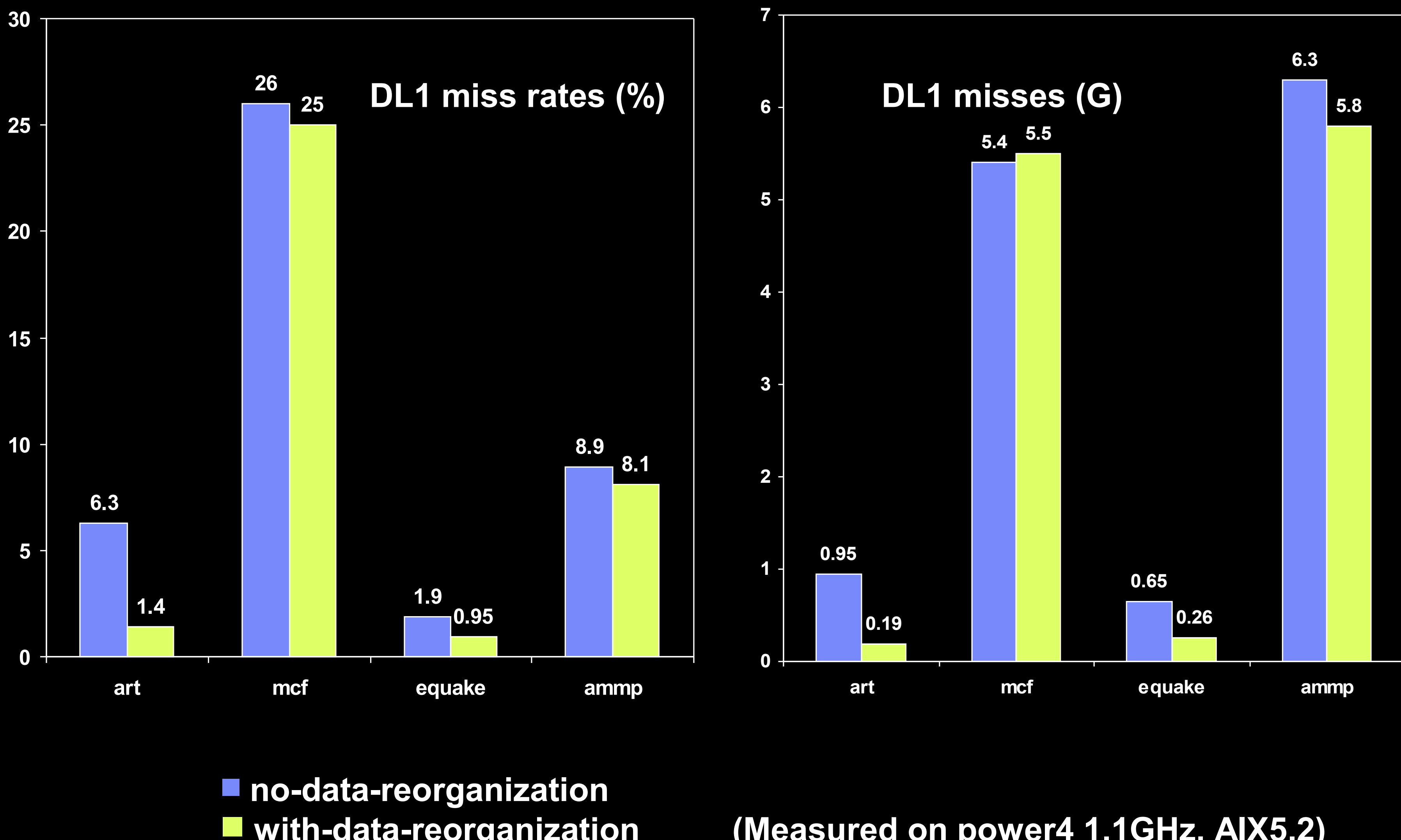
SPEC2000 Performance Improvement With Data Reorganizations



SPEC2000 Performance Improvements With Data Reorganizations

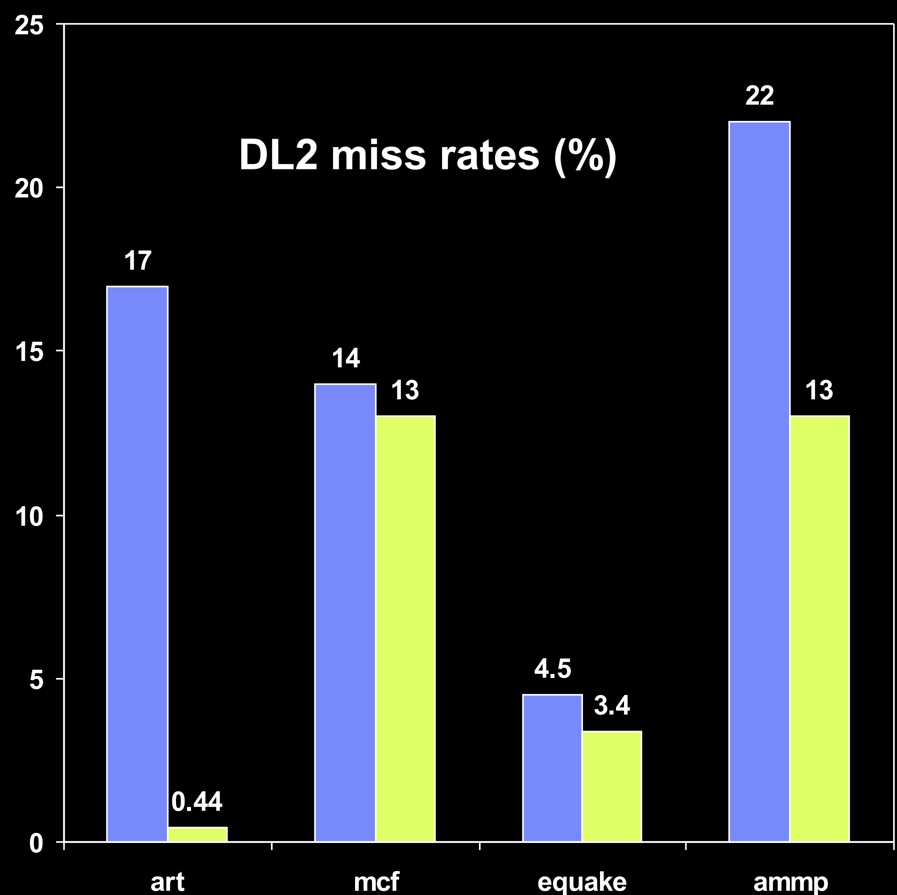


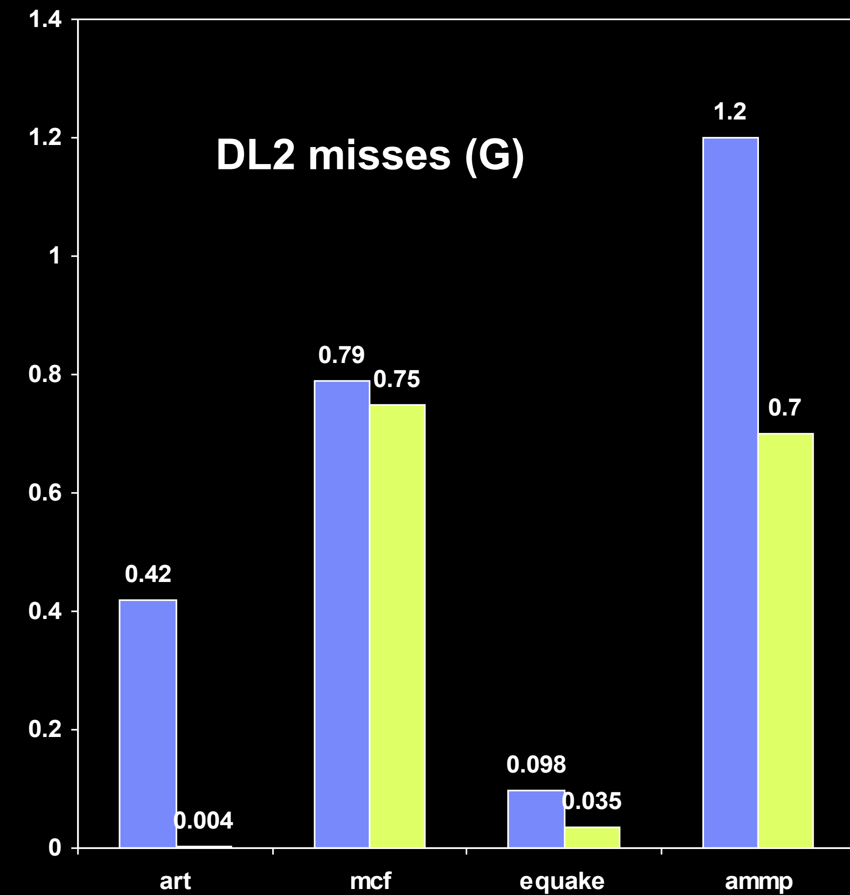
Effect of Data Reorganizations Reduction on DL1 misses



(Measured on power4 1.1GHz, AIX5.2)

Effect of Data Reorganizations Reduction on DL2 misses





- no-data-reorganization
- with-data-reorganization

(Measured on power4 1.1GHz, AIX5.2)

Summary

- A practical framework that guarantees safe automatic data reorganization.
  - Implemented in IBM XL compiler
- Impressive performance improvements on benchmarks and customer codes.
  - Four SPEC2000 benchmarks improved significantly.
- Future work
  - Improve the data shape analysis to capture more complex data access pattern
  - Pursue more data reorganization techniques

Backups

Data Interleaving

- Group data with high affinity and put them together in memory
- Reduce the number of hardware streams and also reduce the cache conflicts
- Target to data in a program region with too many streams.

```
double  a[1000];
double  b[1000];


for (i=0; i<N; i++) {
  … = a[i] …;
  … = b[i] …;
}
```

→

```
struct {
   double  x;
   double  y;
 } ab[1000];

for (i=0; i<N; i++) {
  … = ab[i].x …;
  … = ab[i].y …;
}
```

Data Padding and Alignment

- Array splitting
  - Inter array padding can be added between those new arrays for alignment (e.g., to ensure SIMD alignment), to avoid false sharing.
- Memory allocation merging
  - Intra array padding can be incorporated easily into the framework to avoid cache conflicts

