Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Ian Lance Taylor, *Google*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Struct-reorg: current status and future perspectives

Olga Golovanevsky Ayal Zaks
IBM Haifa Labs—HiPEAC Member
{olga,zaks}@il.ibm.com

Abstract

In this paper we present the ongoing effort to implement C structure optimizations in GCC, its current status and future plans. The general idea of this set of optimizations is to adapt the layout of a data-structure to its access patterns in order to better utilize the cache by increasing spatial locality. These optimizations are known to have many variations and require wide variety of data to be analyzed. An initial implementation of these optimizations in GCC was presented at the GCC Developers' Summit of 2005. Since then many changes took place, including updating to *Tree-SSA* and preparation for GCC mainline 4.3. Additional opportunities were revealed after close analysis of *mcf* benchmark from Spec2000/Spec2006.[1] Struct-reorg relies on careful type escape analysis capabilities, which were provided by K. Zadeck and are now being enhanced to really utilize the *Tree-SSA* form. In addition, as this type of optimization requires the scope of whole program to be analyzed, it is a natural candidate to leverage the *LTO* effort. To prepare its transition to this level, we present optimization flow of data collection stage of struct-reorg optimizations from an *LTO* perspective.

1 Current Status

The effort to implement structure-related data layout optimizations in GCC started in September 2004 as a joint project by Mostafa Hagog and Caroline Tice. Their paper "Cache Aware Data Layout Reorganization Optimization in GCC" [8], presented at the GCC summit of 2005, showed an initial implementation of peeling, splitting and reordering transformations. Since then many changes took place, which we describe in this paper.

The scope of structure reorganization analysis and transformation requires a global view of the program, i.e. as wide as the compiler can be provided. For this reason our implementation leverages the interprocedural analysis (*ipa*) infrastructure developed by Jan Hubička [9], extensively utilizing its call-graph facilities. The struct-reorg-branch, where the project is maintained, was therefore derived from ipa-branch (formerly tree-profiling branch), from which it was constantly updated. Along with other *ipa* optimizations and *ipa* infrastructure itself, struct-reorg optimizations have recently undergone the conversion from *GIMPLE* to *Tree-SSA*. In addition, the flow of its transformation stage was changed from "do something for all functions" mode to "do everything for one function" mode, to correspond to *ipa* infrastructure requirements. After the integration of *ipa* infrastructure (in *Tree-SSA* form) into current GCC mainline (to be version 4.3), and the retirement of ipa-branch, the struct-reorg-branch was aligned with current mainline.

The struct-reorg optimizations use type-escape analysis, developed by Kenneth Zadeck, to guarantee the safety of the transformations. This analysis is also based on *ipa* infrastructure, comprising a separate *ipa* pass, called pass_ipa_type_escape, that precedes struct-reorg optimizations in execution. Along with other *ipa* passes, it evolved from *GIMPLE* to *Tree-SSA* form. Apart from formal transition, the analysis can benefit from the usage of *Tree-SSA* flow-sensitive qualities. Thus we enhanced it to use *Tree-SSA* specific feature for the cases relevant to struct-reorg optimizations, amply described in Section 4.

The efficiency of struct-reorg optimizations strongly depends on the scope of the program available for analysis. The bigger the portion of the program, or compilation unit available, the higher the probability that the structure under consideration does not escape it, so that the transformation can be applied safely. Although the current *ipa* infrastructure along with "combine" and "whole-program" flags can successfully imitate the view of whole program (when all parts of an

[1] In the context of this paper *mcf* benchmark equally represents both 181.mcf benchmark from Spec2000 and 429.mcf from Spec2006

application are compiled together under one compilation command), this solution does not scale for real world applications. The natural intention is to leverage the Link Time Optimizations (*LTO*) infrastructure for struct-reorg optimizations. We believe that the transition of struct-reorg to the *LTO* framework would require additional consideration and revision of struct-reorg's optimization flow to better correspond to the *LTO* infrastructure. In Section 5 we analyze data collection stage of struct-reorg's optimization flow from this perspective.

In addition to the variety of transformations already developed as part of struct-reorg, including: full structure decomposition (*peeling*), hierarchical structure decomposition (*splitting*), and fields reordering (*reordering*), there exist further opportunities detected for the *mcf* benchmark (of Spec2000 [1] and Spec2006 [2]). This new aggressive variant of struct-reorg optimization showed good potential on this benchmark, especially when combined with other GCC optimizations. It is currently undergoing definition, and is expected to be incorporated into the set of struct-reorg optimizations on struct-reorg-branch. The Subsection 3.4 shows this new type of transformation.

Finally, one type of struct-reorg transformation—full structure decomposition, called peeling—is currently being prepared for submission to GCC mainline (to be version 3.4). All the struct-reorg activities, including structure splitting and reordering transformations, the development of new types of transformations like the one inspired by the *mcf* benchmark, as well as experiments and tuning of the decision-making algorithm are hosted on the struct-reorg-branch.

The rest of this paper goes as follows: Section 2 consider data layout optimization and their compatibility with other optimizations in GCC; Section 3 is an overview of the existing transformations, where Subsection 3.4 shows new type of optimization inspired by analyzing *mcf* benchmark; Section 4 presents analysis that provides safety of data-layout transformations and recent extensions made to them; in Section 5 we look inside the optimization itself, analyzing its optimization flow as a potential for joining the *LTO* framework.

2 Data Layout Optimizations in GCC

The idea of these optimizations springs from observing that the allocation of data in a program usually reflects such aspects of programming as its objective orientation, easiness for human perception, readability, modularity, reusability, and more. However the efficiency of program execution is not always on the list, causing data structures to be allocated in a way disregarding the efficiency of their accesses in the program. It is traditionally left to automatic tools, like compilers, post-list optimizers and performance tools, to repair the mismatch between the way data is allocated and the ways it is accessed. Apparently, two ways can be suggested to fix the problem: one is optimizing the accesses to better fit the data layout, the other is selecting the data layout to optimally correspond to the data access patterns.

The layout changing optimizations currently in place in GCC are matrix flattening and transposing, developed by Razya Ladelsky [10], and struct-reorg optimizations presented in this paper. Manipulating with different types of data structures, these optimizations can be applied simultaneously on applications which make extensive use of structures and matrixes. Thus the performance boost of about x5 was measured on 179.art benchmark from Spec2000, when matrix flattening, transposing and structure peeling has been applied on it. This result essentially better than each one, and even the sum of the results, showed by independent application of these optimizations (47% for structure peeling and x2 for matrix flattening and transposing).

Among optimizations changing access patterns are linear loop transformations, such as loop interchange (contributed by Daniel Berlin) loop fusion and fission (under development by Sebastian Pop [14]).

Although both types of optimizations—changing data layout and changing access patterns—can potentially interfere, like matrix transposing with loop interchange, or structure peeling with fusion, they can coexist in synergy. The preferable type of optimization can be selected to better fit the program/dataset, or they can be executed sequentially.

The loop linear optimizations are more efficient at handling multiple loop nests with contradicting accesses patterns, by optimizing each loop separately. When there is commonality among the majority of assesses in the program, and loop nests cannot be interchanged due to dependencies or other constraints, a global change of the data layout is preferable. In addition, when executed sequentially, local optimizations can be applied on "minority" of the accesses, after its majority is treated by

global optimizations. Furthermore, in order to better select the set of optimizations to work on the program/data set, a mechanism for automatic selection of optimal set of optimizations can be used, like the one presented by Grigori Fursin on this summit [7].

3 Struct Reorg Optimizations

This section describes tree types of structure transformations implemented on struct-reorg-branch: full structure decomposition into separate fields (called "structure *peeling*"), hierarchical structure decomposition which partitions a structure into substructures connected by pointers to preserve the unity of original structure (called "structure *splitting*"), and fields reordering (called "structure *reordering*"). Subsections 3.1, 3.2, and 3.3 explain them using examples. A new type of transformation, inspired by analyzing *mcf* benchmark, that combine full structure decomposition with substitution of pointers by indexes, is illustrated in Subsection 3.4.

A decision making algorithm has been developed to prioritize among the different struct-reorg transformations. It is based on control flow graph (*CFG*) with associated basic-block and edge profiling information. First, for each global structure in the program a Fields Reference Graph (*FRG*) is build, with vertices representing individual structure field accesses and edges that follow the *CFG*. The edges of the *FRG* are weighted with the amount of data accessed between pairs of field accesses. The weight information is provided through profiling. Second, the *FRG* is used to build a Close Proximity Graph (*CPG*), which has one vertex for each field of the structure under consideration, and whose edges represent the average amount of data accessed temporally between two fields [8]. Another work on data layout optimizations was found to has a similar approach for analyzing data access patterns in the program [6].

3.1 Full structure decomposition—peeling

Let us suppose there is a structure type definition `str` with two fields—`a` and `b`:

```
typedef stuct {int a; float b;} str;
```

and that an array `arr` of structures of type `str` is allocated through a call to `malloc`:

```
str *arr=(str *)malloc(N*sizeof(str));
```

Let us also suppose that in the program there are two functions `foo_1()` and `foo_2()` which access the `a` and the `b` fields of various `arr` elements in two separate loops:

```
foo_1() {
  ...
  for (i=0; i < N; i++)
    arr[i].a = ...;
  ...
}


foo_2() {
  ...
  for (i=0; i < N; i++)
    ... = arr[i].b;
  ...
}
```

If there are no other accesses to the `arr` array in the program, it would be efficient to change the data allocation so that both loops will access contiguous memory locations.

To cause this change we split the structure into two: one with the single `a` field, and the other with the single `b` field:

```
typedef struct {int a;} str_1;
typedef struct {float b;} str_2;
```

The allocation site of the array is also replaced by two allocation statements:

```
str_1 *arr_1=
     (str_1 *)malloc(N*sizeof(str_1));
str_2 *arr_2=
     (str_2 *)malloc(N*sizeof(str_2));
```

and the access sites in the loops are changed to use the new arrays:

```
foo_1() {
  ...
  for (i=0; i < N; i++)
    arr_1[i].a = ...;
  ...
}
```

```
foo_2() {
  ...
  for (i=0; i < N; i++)
    ... = arr_2[i].b;
  ...
}
```

Figures 1 and 2 illustrate the memory layout of `arr` array before and after the transformation.

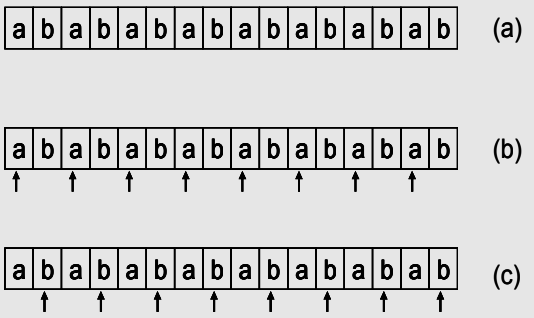

Figure 1: Original memory layout of `arr` array (a), its accesses in function `foo_1()` (b) and accesses in function `foo_2()` (c).

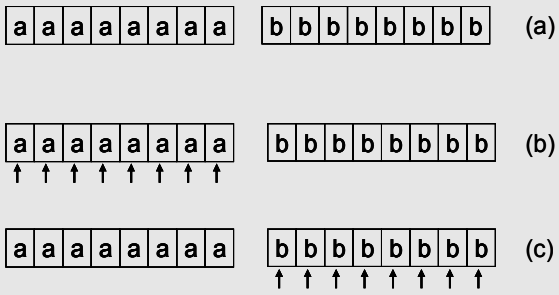

Figure 2: Memory layout of `arr` array after peeling transformation (a), its accesses in function `foo_1()` (b) and accesses in function `foo_2()` (c).

Note that the *struct* itself can be eliminated for single-field structures, replacing the array allocation sites by allocating arrays of fields (integers and floats):

```
int *arr_1=
     (int *)malloc(N*sizeof(int));
float *arr_2=
     (float *)malloc(N*sizeof(float));
```

thereby also simplifying the array accesses in functions `foo_1()` and `foo_2()`:

```
foo_1() {
  ...
  for (i=0; i < N; i++)
    arr_1[i] = ...;
  ...
}
```

```
foo_2() {
  ...
  for (i=0; i < N; i++)
    ... = arr_2[i];
  ...
}
```

This semantical simplification can be important for optimizations running after struct-reorg optimizations. However, we found that this change had an insignificant impact on performance.

Structure peeling however is not always possible. For example, for data structures interconnected by pointers, like link lists and trees, connecting pointers cannot be separated from the rest of the data in the struct. In this case the structure can still be decomposed using additional pointers to keep its original unity, as described next.

3.2 Hierarchical Structure decomposition—splitting

Suppose we have the following Node structure:

```
typedet struct node_struct {
    struct node_struct *parent;
    Expression          left;
    Operator *          bin_op;
    Expression          right;
} Node;
```

it can be decomposed into three separate structures `Node_0`, `Node_1`, `Node_2` as follows:

```
typedef struct {
    Expression   right;
} Node_0;
```

```
typedef struct {
    Expression   left;
    Operator *   bin\_op
} Node_1;
```

```
typedef struct   {
    struct node_struct *parent;
} Node_2;
```

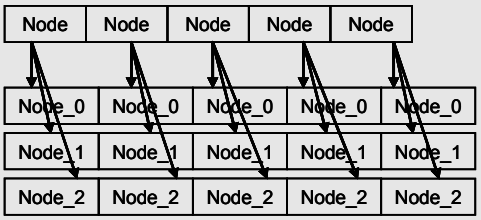

Figure 3: Memory layout of `arr` array after splitting transformation

while redefining the original structure `Node` in the following way:

```
typedef struct node_struct {
Node_0 * Node_0_ptr;
Node_1 * Node_1_ptr;
Node_2 * Node_2_ptr;
} Node;
```

Thus if the original allocation of an array of `Node`'s was:

```
Node *arr=
   (Node *)malloc(N*sizeof(Node));
```

we now allocate additional arrays of types `Node_0`, `Node_1`, `Node_2`:

```
Node_0 *arr_0=
   (Node_0 *)malloc(N*sizeof(Node_0));
Node_1 *arr_1=
   (Node_1 *)malloc(N*sizeof(Node_1));
Node_2 *arr_2=
   (Node_2 *)malloc(N*sizeof(Node_2));
```

and connect them through the pointers to `arr` array:

```
for (i=0; i < N; i++)
{
  arr[i]. Node_0_ptr = &arr_0[i];
  arr[i]. Node_1_ptr = &arr_1[i];
  arr[i]. Node_2_ptr = &arr_2[i];
}
```

Figure 3 shows data allocation of array `arr` after splitting transformation.

Similarly, each access `arr[i].left` to the field `left` of element `i` of the array `arr` (of original `Node` structure) will be replaced by the access `arr[i].Node_1_ptr->left`.

It is worth mentioning that although structure splitting improves cache locality, it is not always beneficial. We have seen that additional memory accesses and indirections, caused by the introduction of connecting pointers, can balance or even overweight the effect of improving cache locality. A well-tuned decision-making algorithm is therefore required to control this transformation efficiently.

3.3 Fields Reordering

An additional type of structure transformations is fields reordering. This transformation changes the order of fields in the structure to correspond to program access patterns. For example, in an array of large structures that undergoes initialization or comparison cycles, it would be preferable to order the fields according to the order in which they are initialized or compared. This type of transformation is less aggressive than structure peeling and splitting—only the definition of the structure type is modified; the allocation and access sites are left intact.

3.4 Peeling plus Indexing

Let us suppose there is self-pointing structure type `node_t`, defining a constructive unit for such interconnected data structures as trees, graphs, link lists, etc.

```
typedef struct node {
  struct node *next;
  int a;
  float b;
} node_t;
```

To instantiate such complex data structure, the array of `node_t`'s is usually allocated:

```
int n;
node_t *arr=
  (node_t *)malloc(n, sizeof(node_t));
```

A typical traversal over this complex data structure access elements of `arr` array through connecting pointers, i.e. in semantically unpredictable order. However different informative fields can be used in different traversals. For example, the following *while* loop access only `a` fields of the `node_t` structure, leaving `b` fields aside:

```
node_t *tmp_node = arr;
while(tmp_node)
{
    tmp_node->a = ...;
    tmp_node=tmp_node->next;
}
```

Although the order of accessing of `a` fields in this loop is unpredictable (`tmp_node = tmp_node->next;`), it's still possible to improve spacial locality by allocating `a` fields together.

Note that structure peeling transformation, described in Subsection 3.1, is not applicable in this case due to self-pointers. The structure splitting (Subsection 3.2) is possible, but insert additional indirections, that influence performance.

Thus to get the desired result, we peel the original structure into separate fields, so that allocation of array of original structures is reorganized to contain contiguous allocations for each field separately, while replacing connecting pointers by indexes in these arrays. In terms of our example, the structure type `node_t` is replaced by set of structure types:

```
typedef struct node_next {
  int next_index;
} node_next_t;
typedef struct node_a {
  double a;
} node_a_t;
typedef struct node_b {
  double b;
}node_b_t;
```

The allocation site of `arr` array is redefined to allocate one chunk of memory virtually partitioned into separate arrays:

```
int n;
size = sizeof(node_next_t)+
       sizeof(node_a_t)+
       sizeof(node_b_t);

node_t *arr=
  (node_t *)malloc(n, size);


node_next_t *arr_next =
          (node_next_t *)arr;
node_a_t *arr_a=
          (node_a_t *)(arr_next+n);
node_b_t *arr_b=
          (node_b_t *)(arr_a+n);
```

The pointers to array bases `arr_next`, `arr_b` and `arr_b` in conjunction with `next_index` indexes are used to access array elements. Thus the *while* loop from our example gets the following form:

```
int tmp=0;
while (tmp!=-1) {
  arr_a[tmp].a = ...;
  tmp = arr_next[tmp].next;
}
```

Note that number of possibilities exist to define mapping between pointers and array indexes. For example, pointers to array bases can correspond to index zero. In this case, the typical comparison of pointer to be equal to zero, can be replaced by comparison to some non legal index value, like, for example, `-1`. Alternatively, a dummy element can be allocated for each array to bear zero value.

The papers [13], [15] exploit the similar approach.

4 Type Escape Analysis Extensions

Initially written for aliasing [5], the type-escape analysis engine (ipa-type-escape.[c,h]) provides safety analysis for struct-reorg optimizations. It guarantees that all instances of a given structure type do not escape the compilation unit, and are used "conventionally" inside it.

A structure type is said to escape the compilation unit, if a pointer to the structure is an actual parameter or return value of a global function (accessible from outside the compilation unit), or the structure is instantiated as a global variable, or one of its fields is known to escape.

A structure type is considered to be used unconventionally and cannot be transformed safely, if a pointer to the structure undergoes casting or if operations other than plus, minus or multiplication are applied to it.

The type-escape analysis algorithm is comprised of two phases, initially written in *GIMPLE*. First, all functions are traversed statement-by-statement, collecting information on explicitly escaping types. Then a transitive closure is executed over sub- and super-structures of the escaping types. During the first analysis phase a strict policy was initially applied due to *GIMPLE* representation: if a statement does not provide sufficient information to determine the purity of the type, it was considered to be escaping. With the conversions of *ipa* to

Tree-SSA form, it became possible to use flow sensitive information to relax this limitation.

For example, suppose there is a structure type `str_t`:

```
typedef struct {
  int c;
  double b;
} str_t;
```

and an array of `str_t` structures, allocated by a call to `malloc`:

```
str_t *a_p =
   (str_t *)malloc(N*sizeof(str_t));
```

to be later accessed within a loop as follows:

```
for (i=0; i < N; i++)
  a[i].c = 5;
```

Then the following statements will be generated as part of translating this loop into *Tree-SSA*:

```
  i.0_3 = (unsigned int) i_1;
  D.1603_4 = i.0_3 * 16;
  D.1604_5 = (struct str_t *) D.1603_4;
  a_p.1_6 = a_p;
  D.1606_7 = D.1604_5 + a_p.1_6;
  D.1606_7->c = 5;
```

When the first phase of ipa-type-escape analysis reaches the 5*th* statement in this list, it cannot determine without investigating use-def chains of variables `D.1604_5` and `a_p.1_6` that these two variables are actually the base and offset of an array allocated by a call to `malloc`. Therefore, having a summation of two pointers to struct `str_t`, the conservative assumption was taken that the `str_t` type escapes.

To relax the case of adding pointers to structures, the analysis was enhanced to recognize the case when these pointers are actually base and offset of an array, with the function *is_array_access_trough_pointer_and_index* as its entry point. This function checks whether one of the pointers passed one casting operation from non-pointer type and is actually generated by multiplication of index by size of the structure, while the other was never casted. The *Tree-SSA* interface function *walk_use_def_chains* is used in implementation from callback of which the *walk_tree* function is executed for each *def_stmt*.

This enhancement enabled for example structure `f1_neuron` from 179.art benchmark from Spec2000 to be identified as non-escaping by ipa-type-escape analysis, thereby permitting struct-reorg transformations to be applied to it safely. The increase in compilation time caused by the added analysis measured on this benchmark was between 0 and 1% of total compilation time.

Additional contributions to ipa-type-escape analysis include:

- relaxing the case of adding a constant to a pointer. For example, in term of previous example a field `c` of array `a_p` is accessed as `a_p[5].c`
- allow multiplication of pointers (suggested by Jan Hubička).
- recognizing casting from pointer type as escaping. For example, if there is a structure `str_t` as it was previously defined, an array `str_t a[N]` and the following piece of code:

  ```
  void *vp = (void *) &(a[0].c);
  str_t *a_p = (str_t *)vp;
  ```

  then the cast `(str_t *)vp` is recognized as causing structure type `str_t` to escape.
- recognizing calls to `malloc()` after analyzing through casts. For example, the following piece of code:

  ```
  #define N 1000
  str_t *a_p =
   (str_t *)malloc(N*sizeof(str_t));
  ```

  is translated into *Tree-SSA* form as follows

  ```
  D.2198_2=malloc (16000);
  D.2199_3=(struct str_t *)D.2198_2;
  ```

  Thus when analyzing the second statement in this list, it's possible to determine that `D.2198_2` is the result of `malloc` function by using its use-def chain.

5 Optimization Flow Analysis

When an optimization is considered a candidate for *LTO* [4], it is important to keep in mind the characteristics of this infrastructure. In the *LTO* compilation model, each file, or part of the program, is first compiled separately. At the *GIMPLE* intermediate representation level the program is serialized into an object file, bypassing the

rest of the compilation cycle. When all parts of the program have passed this first compilation stage, they are compiled together. This second compilation stage resumes from the point at which the first compilation stopped. It works on the *GIMPLE* representation of the whole program, except libraries, resurrected from the object files. Thus optimizations executed at the second stage of compilation benefit from the whole program scope of compilation.

The second stage of the *LTO* compilation model can therefore be extremely beneficial for struct-reorg optimizations, providing a whole program view for their analysis. The struct-reorg optimizations strongly depend on type-escape analysis. If a structure escapes the compilation unit, none of the struct-reorg transformations can be applied to it. Hence the wider the scope of program view provided for these optimizations, the more powerful they can be.

The benefit of providing whole program view, however, has its price: memory consumption is increased as well as compile time. Memory consumption can be reduced by reducing the size of intermediate representations (*IR*). Such works as "Memory SSA" by Diego Novillo [11], "GIMPLE Tuples" [12], and more [3] target this factor. The amount of data collected by each optimization can also be influential. Compile time can be improved through parallelization. For example, both the data collection and transformation stages of optimizations can be parallelized in a 'per function' manner, in case they are independent.

The struct-reorg optimizations are built up of tree optimization stages, similar to many other compiler optimizations. First stage is responsible for data collection required by the optimizations. At the second stage the collected data undergoes an analysis process in order to decide which of the existing struct-reorg transformations is most effective. Finally, the third stage applies the transformation.

In the rest of this section we consider current implementation of data collection stage of struct-reorg optimizations, and suggest changes that can be beneficial for it in order to be incorporated into the *LTO* infrastructure.

Figure 4 shows the existing flow of data collection stage, where FRG(*str*) is FRG for structure *str* and implementation of the function `build_f_acc_list_for_bb(bb, str)`. As a result of data collecting, FRG's are built for all structures in the program. To allow the parallelization of the data collection stage on a 'per function' basis, we introduce FRG for each function separately to prevent accessing of global data. Then the data collection flow can be implemented by interchanging the outermost loop with the loop enclosed in it (Figure 5).

```
for each structure str in the program
  for each function func in the program
    for each basic block bb in func
      /* Build the list of accesses to fields
         of str that appear in bb. */
      list = build_f_acc_list_for_bb (bb, str);
    /* Update FRG(str) with accesses to str in the func. */
    for each basic block bb in func
      update_FRG(str) with list
    destroy all lists of func

build_f_acc_list_for_bb (bb, str)
{
  initialize the list list of accesses of str in bb
  dist = 0;
  for each stmt in bb
    {
      if (stmt accesses a field of str)
      {
        add access to this field to the list with dist
        dist = 0;
      }
      else if (stmt accesses memory)
        dist += size_of_accessed_data
    }
}
```

Figure 4: Flow of data collection stage

However, even with this change, each function in the program is traversed repeatedly, i.e. per each data structure in the program. The data collection flow can be further changed so that each function gathers all data required by optimization through one traversal. Figure 6 shows this change.

6 Conclusions

In this paper we considered bundle of structure reorganization optimizations developed on the base of GCC compiler. Four types of structure transformations has

```
for each function func in the program
  for each structure str in the program
    for each basic block bb in func
      /* Build the list of accesses to fields of str
         that appear in bb.  */
      list = build_f_acc_list_for_bb (bb, str);
    /* Update FRG(str,func) with accesses to str
       in the func.  */
    for each basic block bb in func
      update_FRG(str, func) with list
    destroy all lists of func
for each structure str in program
  for each function func in program
    update FRG(str) with FRG(str, func)
```

Figure 5: Parallelized data collection algorithm

been presented—full structure decomposition (*peeling*), hierarchical structure decomposition (*splitting*), fields reordering (*reordering*) and *peeling* plus indexing—among which the former three are already developed on struct-reorg branch, and the latter, which was found to be effective for data structures interconnected by pointers, similarly to one presented in *mcf*, is now under development. The comparisons between these and other GCC optimizations working on the spacial locality (like liner loop transformations) showed the complementary character of these optimizations. They may be effective either when applied in conjunction or when preferable type of optimization is selected.

Naturally the applicability of struct-reorg optimizations is restrained by the scope of the program available for analysis. On one hand, the type-escape analysis stands to guards the safety of these optimizations. We have shown how the strictness of type-escape analysis can be relinquished through extensive use of *Tree-SSA IR*. On the other hand, the *LTO* infrastructure can be utilized to provide whole program view for these optimizations. To be based on this infrastructure, the optimization flow/memory consumption has to be adapted to be optimal for its needs. Thus possible enhancements to the flow of struct-reorg data collection stage have been presented as example of this adaptation effort.

Finally, since wide variety of applications and data sets is essential for effective tuning of both decision making mechanism and type-escape analysis, we'll focus on this activity in a close future.

```
for each function func in the program
  for each basic block bb in func
    /* Build the list list of accesses to fields
       of all structures in the program.*/
    list = build_f_acc_list_for_bb (bb);
  /* Update FRG(func) with accesses to
     all structures in the func.  */
  for each basic block bb in func
    update_FRG(func) with list
  destroy all lists of func
for each structure str in program
  for each function func in program
    {
      reduce FRG(str, func) from FRG(func)
      update FRG(str) with FRG(str, func)
    }

build_f_acc_list_for_bb (bb)
{
 /* Initialize the list list of accesses in bb
 to fields of all structures in the program. */
 init list
 dist = 0;

 for each stmt in bb
   {
    if (stmt accesses any field of any str in the
program)
      {
       add access to this field to the list with dist
       dist = 0;
      }
    else if (stmt accesses memory)
      dist += size_of_accessed_data
   }
}
```

Figure 6: Parallelized data collection algorithm with only one traversal for each function

References

[1] SPEC CPU2000. `http://www.spec.org/cpu2000/`.

[2] SPEC CPU2006. `http://www.spec.org/cpu2006/`.

[3] Speedup area on GCC Wiki. `http://gcc.gnu.org/wiki/Speedup_areas`.

[4] Link-Time Opmitization in GCC: Requirements and High Level Design, November 2005. `http://www.spec.org/cpu2000/`.

[5] Daniel Berlin and Kenneth Zadeck. Compilation Wide Aliasing Analysis. In *Proceedings of the GCC Developers' Summit*, pages 15–24, Ottawa, Canada, June 2005.

[6] E.Raman, R.Hundt, S.Mannarswamy. Structure Layout Optimization for Multithreaded Programs. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, San Jose, California, March 2007.

[7] G.Fursin, C.Miranda, S.Pop, A.Cohen, O.Temam. Practical run-time adaptation with procedure cloning. In *Proceedings of the GCC Developers' Summit*, Ottawa, Canada, July 2007.

[8] Mostaga Hagog and Caroline Tice. Cache Aware Data Layout Reorganization Optimization in GCC. In *Proceedings of the GCC Developers' Summit*, pages 69–92, Ottawa, Canada, June 2005.

[9] Jan Hubicka. Interprocedural optimization on function local SSA form in GCC. In *Proceedings of the GCC Developers' Summit*, pages 75–84, Ottawa, Canada, June 2006.

[10] Razya Ladelsky. Matrix flattening and transposing in GCC. In *Proceedings of the GCC Developers' Summit*, pages 97–108, Ottawa, Canada, June 2006.

[11] Diego Novillo. Memory SSA. In *Proceedings of the GCC Developers' Summit*, Ottawa, Canada, July 2007.

[12] R.Henderson, A.Hernandez, A.Macleod, D.Novillo. Tuple Representation for GIMPLE. `http://gcc.gnu.org/wiki/tuples?action=AttachFile&do=get&target=tuples.pdf`.

[13] Rodric M. Rabbah and Krishna V. Palvem. Data Remapping for Design Space Optimization of Embedded Memory Systems. *ACM Transactions in Embedded Computing Systems*, 2(2):186–218, May 2003.

[14] S.Pop, A.Cohen, C.Bastoul, S.Girbal, C.A.Silber and N.Vasilache. GRAPHITE:Polyhedral Analyses and Optimizations for GCC. In *Proceedings of the GCC Developers' Summit*, pages 179–198, Ottawa, Canada, June 2006.

[15] Y.Zhong, M.Orlovich, X.Shen, C.Ding. Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity. *PLDI*, 39(6):255–266, 2004.