Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

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

Cache Aware Data Layout Reorganization Optimization in GCC

Mostafa Hagog
IBM Research Lab in Haifa
mustafa@il.ibm.com
Caroline Tice
Apple Computer, Inc.
ctice@apple.com

Abstract

Many programs suffer from poor cache behavior due to poor locality of their data. We are interested in optimizations that improve data locality by changing the layout of data structures in a cache aware manner. One technique for changing the layout of a data structure is to change its high level definition (*struct* definitions in C and *struct/class* definitions in C++). There are two such transformations that we can perform on data structures: structure splitting (splitting fields into several sub-structures) and field reordering. In this paper we present our implementation of these optimizations in GCC. Our optimization consists of three main stages: Deciding if it is safe to perform the optimization; Determining how to rearrange the data records; and, applying the transformation. We measured the performance impact of our optimization and discovered that while some programs had no noticeable performance improvements, in at least one case there were significant gains in cache behavior and overall performance.

1 Introduction

General purpose programs with large data sets tend to suffer from high cache miss ratios and thus from a severe degradation in performance. The cache memory in particular, and the memory hierarchy system in general, was introduced by computer architects several decades ago based on the observations that applications tend to follow the locality of reference principle. This principle states that programs tend to spend 90% of their execution time accessing 10% of their data. This is also known as the 90/10 rule [6]. There are two types of data locality: *temporal locality* and *spatial locality*. *Temporal locality* states that if an item is referenced, it will tend to be referenced again soon. *Spatial locality* states that items stored in nearby memory locations are likely to be accessed close together in time [5]. Programs which violate the data locality rule exhibit "poor data locality." Programs with poor data locality do not use the memory hierarchy system very efficiently, and hence their performance suffers.

To solve the problem of poor data locality there are two main approaches: The first, which is

widely used and is considered traditional, is to change the access patterns in the program. Such optimizations change the order of instructions accessing the data (e.g. loop interchange, loop distribution, etc.). The other approach is to change the layout of the data itself. We take the latter approach. We attempt to improve cache behavior of general data structures that involve both *structs* in C and *structs* and *classes* in C++. Such an optimization doesn't concentrate on any particular loop; when performed it affects all places in the entire program that use the data type for which the memory layout has changed.

A variety of work has been done in this field. The less aggressive approaches are those that try to reorder data elements: reordering of automatic variables on the stack; reordering of global variables [1]; field reordering within a record [4, 2]; and cache-aware reorganization of heap allocated objects [1, 3]. A more aggressive optimization is structure splitting [4, 2], in which each data record is split into two or more records (sub-structures). In each sub-structure, the fields that are likely to be accessed together all over the program are grouped together.

Throughout this paper we need to distinguish carefully between the definition of a record data type (*struct* or *class* definitions) and the use of such data types to construct larger data structures (trees, graphs, arrays, lists, etc.). Therefore for the rest of this paper we will use the term "record" to refer to an instance of a record definition, and the term "data structure" to refer to larger constructs based upon and built from the individual records. Also, for the sake of brevity, we may sometimes refer to "types" in the source program when we really mean "variables declared to be of the types."

We concentrated on two transformations: field reordering and structure splitting. Any work on implementing these two transformation has to address three main issues: Guaranteeing that the code will be correct after applying the transformation (correctness issue); Deciding what has to be done to achieve the desired layout (transformation issue); and, determining how to we reorder or split the fields to obtain better performance. All of the previous work in this area tacitly agrees that finding the optimum new layout is an NP-complete problem, and even finding an approximation to it is very hard. However several studies have proved that such optimizations can provide significant performance gains in various real world programs. There are also several studies that report improvement on the SPEC2000 benchmark suite and other known benchmarks, using such techniques. So, how do we determine the desired layout? It is common to use a cost-model that decides on the desired layout based on profiling information. In our work we use the existing GCC basic block profiling mechanism to provide an estimation of the access patterns to the data elements (fields of a record).

The basic block profiling in GCC provides us with execution counts for each basic block in the instrumented program. Our optimization takes the control flow graph (CFG) with the counts on the basic blocks, and stores for each block the list of fields accessed within it (for records in which we are interested). It then builds a graph that represents accesses to fields. This graph is then traversed and compacted into another graph that represents relations between fields (rather than accesses to fields). We call this the Close Proximity Graph (CPG), which is described in Section 2.1. We traverse the CPG using various heuristics and algorithms, making recommendations about how to rearrange the fields of the records in question.

Once we have the recommendations, we verify that the desired transformations are safe and legal to perform, with respect to the rest of the program. If so, we apply the transformations. There are three types of transformations

that our optimization can attempt: field reordering, structure splitting and structure peeling. Field reordering is just what it sounds like: we change the order in which fields are defined within the record to match more closely the way they are accessed by the program. Structure splitting is removing some fields from the original record and placing them into a new record, which is then pointed to from the original record. Structure peeling is a special case of structure splitting. Each of these transformations is described in more detail later in this paper.

After implementing this optimization in GCC, we ran it on the SPEC 2000 benchmark suite to see what performance improvements we obtain. While in most cases the performance gains were negligible, for one particular benchmark which had exhibited particularly poor cache performance, we saw a performance gain of 47%. In no case was there a noticeable performance penalty.

The rest of this paper is organized as follows. In Section 2 we give an overview of our implementation. Section 3 describes the various design and implementation issues in greater detail. We present our measurement data in Section 4, and in Section 5 we describe the current state of the implementation and our future plans.

2 Design details

2.1 The Close Proximity Graph

The Close Proximity Graph (CPG) summarizes a relation between each pair of fields called the close proximity (CP) relation. The close proximity relation between two fields is an estimation of the likelihood of accessing one field "shortly" after accessing the other during the program execution (temporal closeness). The estimation of the close proximity is usually based on profiling information. There are several different profiling techniques that were used in previous work. Some collect trace data and others are based on path profiling, while still others are based on basic-block profiling. After the profile information is gathered, the information is collected in a relation graph, which tracks relations between the data elements. This graph is then traversed to determine the desired grouping (splitting) or ordering of the fields. The profiling information could be highly accurate (tracing) or could have poor accuracy. However the fact that heuristics rather than optimal algorithms are used to traverse the relation graphs can mask the difference in profile accuracy. This technique uses the execution counts of the basic blocks to estimate the number of times a pair of instructions are executed "consecutively" during the instrumentation phase (As described in [5]). The CPG summarizes for each pair of fields the counts and "distances" of instruction pairs that accessed the fields. In this case "distance" is the amount of memory accessed in between the execution of the access instructions pair.

As previously mentioned, GCC uses the basic block profiling technique. To use this data we first build what we call the field reference graph (FRG) which is the same as the control flow graph (CFG) except that a node in the FRG represents a memory access and a node in the CFG represents a basic block (see Section 3.2 for more details). After the FRG is built it is traversed to calculate the close proximity relation for field pairs and store it in the CPG. A straightforward way for traversal is to do an exhaustive search over all the FRG paths. The problem is that this traversal has exponential complexity. Instead we implement a different algorithm which we call the FRG collapsing algorithm, and which has linear complexity.

This relies on the fact that basic block profiling is losing some information (compared to tracing) so the inaccuracy in this algorithm doesn't cause additional loss of accuracy—this is described in more detail in Section 3.2.2.

Once we have the CPG we traverse it to determine the recommendations for structure splitting and field reordering. Field reordering is straightforward; field pairs that have high close proximity (according to CPG) are stored in close locations in the record memory layout. Doing so increases the program data locality, because we increase the probability that data which is accessed temporally "close" will be in consecutive memory locations. In Section 3.2.3 we describe the algorithm and heuristic to find the ordering of the record fields that best satisfies the field pairs close proximity relations.

In structure splitting we want to change the layout of the whole data structure by changing its records (through their type definition). We separate the fields that are not likely to be accessed close together temporally into different records (or different sub-records), and fields that are likely to be accessed close in time remain in the same record. This avoids the penalty of bringing irrelevant data into the cache. Often only a few of the fields of a record are accessed most of the time, for example traversing a linked list using only the key and next fields. In this case we separate the key and next fields from the other fields of the linked list record. In the general case, when deciding how to split a record, we deal with two main issues: The first is how to divide the sub-record (which fields should be joined and which must be kept away); the second is the performance tradeoff between wanting to keep fields without close accesses away from each other, but needing an additional pointer dereference for each additional new sub-record (to access the next record). A detailed description of this is in Section 3.2.5.

2.2 Three Types of Data Layout Reorganization

There are three types of data structure layout transformations that our optimization can perform: field reordering, structure splitting and structure peeling. Field reordering is a transformation where we create a new record definition with the same fields, of the same types, with the same names, as in the original record, but with the fields defined in a different order. Figure 1 shows an example of this type of transformation. All occurrences in the program of the old type are then replaced with the new type.

```
struct network {
    char        clustfile[]
    char        inputfile[]
    long int    n;
    long int    n_trips;
    long int    max_m;
    long int    m;
    long int    m_org;
    long int    m_impl;
    long int    primal_unbounded;
    double      optcost;
    long int    ignore_impl;
    node        *nodes;
    node        *stop_nodes;
    arc         *arcs;
    arc         *stop_arcs;
    long int    bound_exchanges;
    long int    checksum;
};
```

Original Struct

```
struct network {
    long int    checksum;
    long int    bound_exchanges;
    long int    ignore_impl;
    double      optcost;
    long int    primal_unbounded;
    long int    m_org;
    char        clustfile[]
    char        inputfile[]
    node        *nodes;
    node        *stop_nodes;
    arc         *stop_arcs;
    long int    max_m;
    arc         *arcs;
    long int    n_trips;
    long int    n;
    long int    m;
    long int    m_impl;
};
```

Reordered Fields

Figure 1: Example of field reordering.

Structure splitting involves pulling some of the fields out of the original record and putting them into one or more new records. Each new record is pointed to by either a new field in the original record, or a field in one of the other new records. Figures 2 and 3 show an example of this type of transformation.

Structure peeling is a special case of structure splitting. As with structure splitting, fields are pulled out of the original record and placed into one or more new records. Unlike structure splitting, the new records do not need to be pointed to by fields within the other records. This leads to slightly more efficient, compact

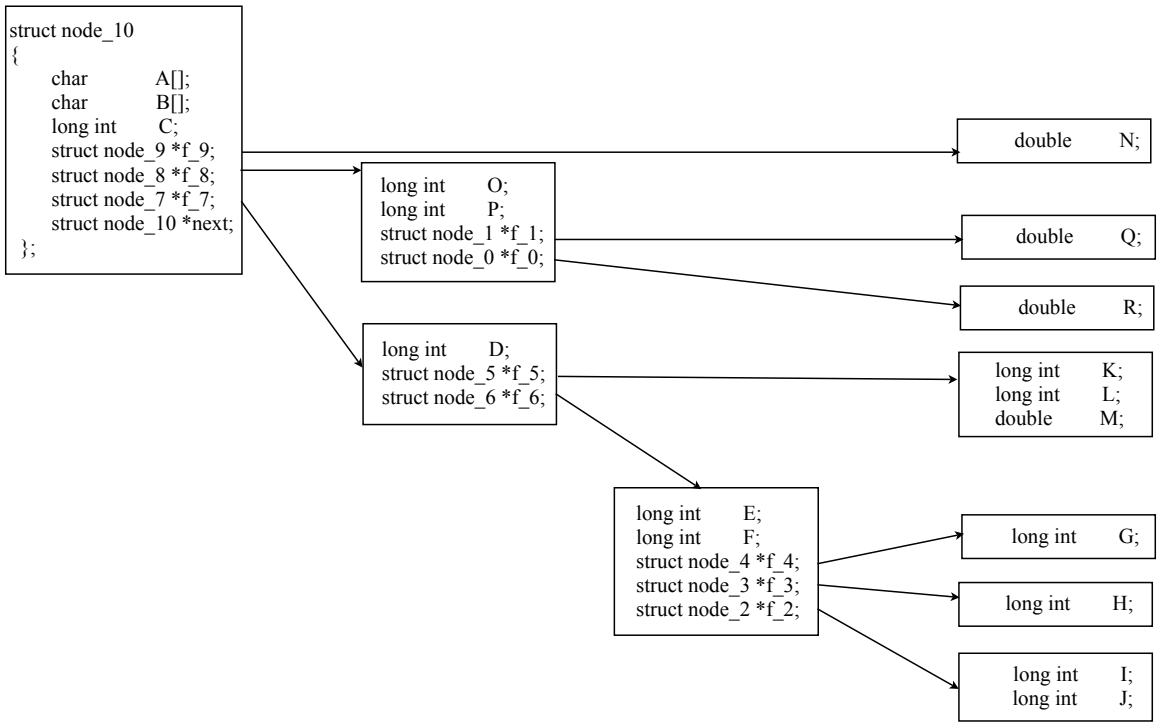


Figure 3: Example of record hierarchy, after structure splitting.

```
struct node
{
    char        A[];
    char        B[];
    long int    C, D, E, F, G;
    long int    H, I, J, K, L;
    double      M;
    double      N;
    long int    O;
    long int    P;
    double      Q;
    double      R;
};
```

Figure 2: Example record definition, before splitting.

records, but is only appropriate in those cases where the original record was never used as part of a dynamic data structure (i.e. it was only used for creating arrays).

one desired goal of our transformations is to reduce the size of the records, therefore structure peeling is preferred over structure splitting wherever possible, as structure splitting requires the addition of one new field for each new record created. However we cannot always perform structure peeling. For example, consider a student information record that contains seven fields: name, address, age, classes, grades, GPA and next. The record is used to create a linked list of student records. Suppose that there are three students, so there are three records of type *student*. After applying the data profiling and analysis, we determine that the fields name, classes, grades and GPA should be together in one record and the fields age and address should be in a separate record.

Now if we attempt to perform structure peeling we have a problem. When we want to access a student's age or address, we have three dangling age-address records to choose from, with nothing pointing to any of them, and no way of telling which age-address record belongs to which student's original record. By comparison, if the student records were being used in an array rather than a dynamic data structure, we could use the array index to match all the new records with the original records (see Figure 4).

2.3 Safety and Legality Issues

As with all optimizations, before we can apply any of these transformations, we need to verify that the transformation we want to apply is both safe and legal for the particular program we are optimizing. In other words we need to be absolutely certain that it will not alter the semantics or behavior of the program in any way.

In order for it to be safe to rearrange the layout of fields within a record or to pull them into a new record, we need to check every field access within the program and make sure the access does not depend on the data within the record being laid out in a particular manner. This is a fairly simple concept, but implementing it involves several not-so-simple issues.

The first problem is finding "every field access within the program." There are two major subproblems implied by this phrase. The first subproblem is that we have to examine the *entire* source program before we can apply *any* transformations.

Once we have made sure we have the entire program to analyze, we still have to ensure that no variable declared to be of a type we are considering splitting is passed as a parameter to an external (library) function (escape analysis).

The next step is to analyze the code that we have. This brings us to the second major subproblem: *recognizing* field accesses as such. Because we are dealing largely with C, which is a very flexible language, there are many different ways in which a field of a record can be accessed: directly by field name; through a pointer to the base variable plus an offset; through a pointer to an adjacent field plus or minus an offset; or through an aliased typecast variable (to name a few methods). A little thought should convince the reader that out of all of these access methods (and others that may come to mind), only the first, directly by field name, is safe for us to transform; all of the others would be broken by changing the data layout. Therefore our analyses check every variable and parameter in the program that has a type of `RECORD_TYPE` (possibly nested in one or more `POINTER_TYPE`s). If any such variable has its address taken and cast to something else, or if its address is used in any pointer arithmetic, we declare the type unsafe to transform. If a type passes all of these checks, we consider it safe and legal to transform.

3 Low-level design details

We divided the implementation of this optimization into three stages: performing analyses to determine which, if any, data records were safe and legal for us to transform (Stage 1); examining the profile information, building the Close Proximity Graph and generating the recommendations for data layout transformations (Stage 2); and applying the recommended transformations to the program (Stage 3). We will discuss each of these stages in more detail below.

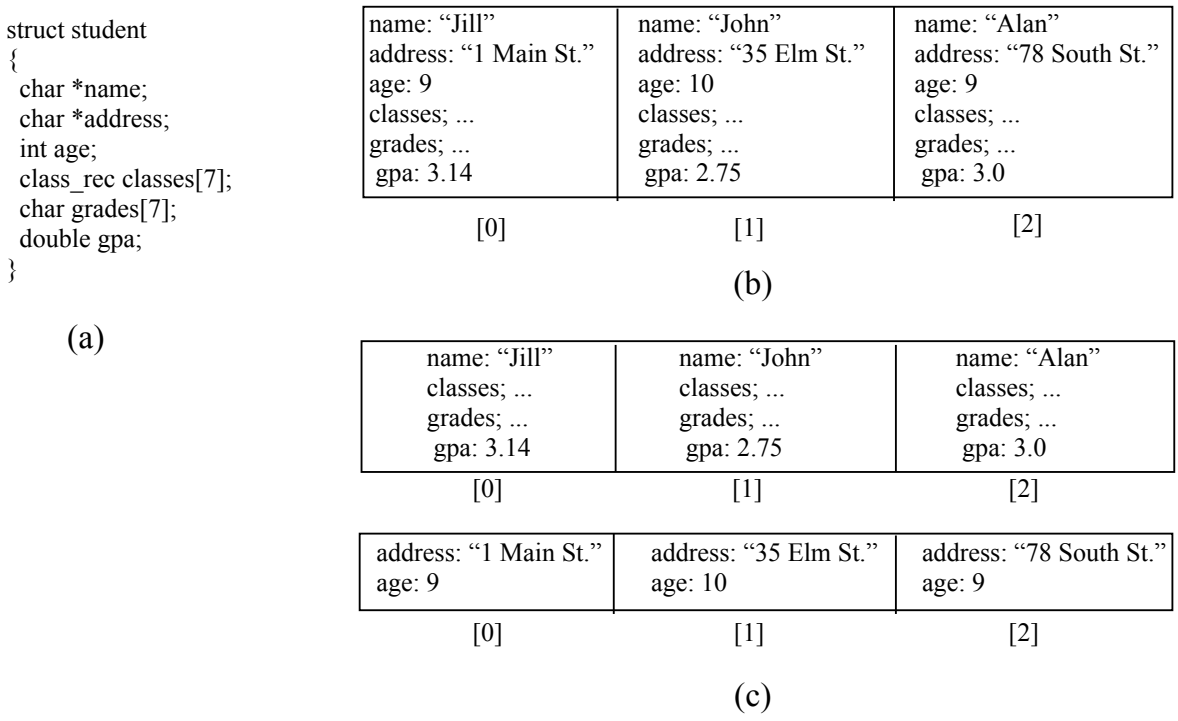


Figure 4: Solving the peeled reference problem for arrays.

3.1 Stage 1: Safety and Legality Analyses

Because we are transforming data types which might be global to the entire source program, we have to verify that the *entire source program* is safe and legal for us to transform, before we apply any transformations. Figure 5 helps to show why this is necessary. Suppose we are considering to apply this optimization to a program that consists of three source files (one header file and two other C files), and suppose we allowed the optimization to be applied without the requirement that we see all the source files at once. As you can see, the header file contains the definition of the record, *bad*, which is used to create a global data structure, *G*, which is accessed by both source files. First we compile `File1.c`, which contains no illegal references to the record type, so we go ahead and perform structure splitting, as recommended. In a separate compilation pass, we compile `File2.c`, where we come across an illegal reference to the record, so we do not perform structure splitting. Next we attempt to link the two `.o` files, which now contain conflicting definitions for the data types. From here on, it is impossible to predict what precise behavior will be encountered, but there is guaranteed to be an error at some point. The error might manifest itself when the linker attempts to link the `.o` files that contain conflicting type definitions (although it is certainly possible for the link step to succeed). The program may crash with a bus error as soon as the user attempts to run it; or it may run for quite a while, either generating random wrong results, or appearing to work correctly for a while and then suddenly failing. The exact nature of the error and point of manifestation will depend substantially on the individual program being compiled. Obviously such nondeterministic and incorrect behavior is completely unacceptable. The only way to guarantee that such "half-splitting" does not occur is to have

the previously stated requirement, that the compiler be given the entire source program at one time. Therefore we have a meta-safety issue that needs to be addressed first, namely we have to ensure that the compiler has been passed all of the user written code for the entire program at once.

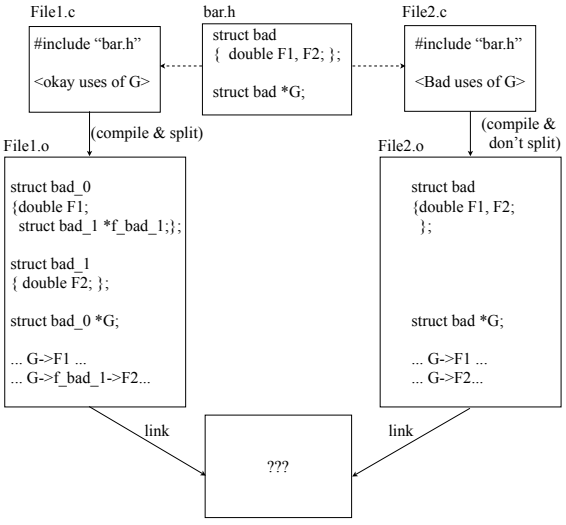


Figure 5: What happens if we do not use the `-fwhole-program` flag.

The difficulty is that it is impossible for the compiler to determine on its own whether or not it has been passed all the user-written code. In particular, calls to non-standard library functions, which are probably "safe" as far as we are concerned (unless one of the arguments is a variable of a type we are interested in), look exactly like calls to non-library functions, which may not be safe. Since the previous statement may not be immediately obvious, let us clarify. A function is "safe" if is does not contain any unsafe references to a field of a record we are interested in rearranging. Presumably the records in question are defined in the program we are compiling, so no library function should touch them unless one is passed to the library function as a parameter. On the other hand, functions that are part of the program we are compiling are freely able to create and access variables of types in which we are interested, so all such functions are by default not safe.

Since it is impossible for the compiler to determine if it has been passed all the user-written code for the entire program, we have settled for a close approximation: to force the error to appear at the earliest detectable moment, namely link time. We use the `-fwhole-program` flag implemented by Jan Hubicka on the tree-profiling-branch to do this. The flag works as follows: If this flag is set, then all functions and global variables passed into the current compilation are marked as *static*, i.e., private to the current compilation. Thus if a program is compiled in multiple parts, either the parts must be entirely disjoint (i.e., neither contains any reference to any function or variable in the other part) or when the program is linked there will be a link-time failure, as pieces from one compilation try to access static pieces from a separate compilation. There are a couple of exceptions to this behavior. The function *main* is, of necessity, left globally visible, and any decl marked as `DECL_EXTERNAL` is left globally visible. Our data layout reorganization optimization then checks to make sure that the `-fwhole-program` flag has been turned on before attempting to do any transformations. Thus we guarantee either that we are passed the entire program to analyze and transform, or that the whole thing will fail at link time.

Having taken care of our meta-safety issue, we still have to verify that no uses of records we wish to transform "escape" to code that we cannot analyze. The escape analysis is performed by traversing the call graph structure and examining every call to a function for which we do not have a body. If any argument to any such function call has a type we are considering transforming, that type is marked as unsafe to optimize.[1]

[1]Function call arguments that are references to indi-

3.1.1 Building on the work of others

The actual analysis of the variables (both the alias analysis and the pointer math checks) are done by the ipa-static-vars-analysis work written by Kenneth Zadeck. Many of the alias and escape questions we needed to answer were the same ones Kenneth Zadeck was working on in his improved alias analysis work, which he did on the tree-profiling-branch. He very helpfully wrote some extra functions to help us obtain the particular information we needed, and ported his work to the struct-reorg-branch. His analyses are all in the file `ipa-static-vars-analysis.c`.

Our work is interprocedural in nature and has to be applied to the entire program before function-specific optimizations are applied, we perform our optimization at the cgraph level, and incorporate much of the work Jan Hubicka, Steven Bosscher, Daniel Berlin and Kenneth Zadeck have been doing on improving the accuracy of the information available at the cgraph nodes, and the quality of the analyses that can be performed on them. This especially includes the `-fwhole-program` flag (mentioned previously) and various modifications necessary to make multi-file programs compile correctly with this flag.

Using the `-fwhole-program` flag implies that we also need to use the `-funit-at-a-time` flag and the `-combine` flag, if the program contains more that one source file, which in turn means we needed to add some code to `c-decl.c` to make it do the right thing when it encountered multiple definitions (resulting from multiple files including the same header files) that were really the same thing. We borrowed heavily from Geoff Keating's work on the apple-ppc-branch to make this work correctly.

vidual fields within a record are okay, as long as the type of the field itself is not based on a `RECORD_TYPE`.

We also augmented the cgraph information to contain accurate information about indirect function calls (calls made through function variables). This was necessary as we needed accurate information about which functions might be called to be absolutely sure that we did not overlook anything when performing the safety and legality checks for our optimization.

3.2 Stage 2: Building and Using the Close Proximity Graph

The CPG summarizes the likelihood of each pair of fields being accessed close together temporally during the program execution. We build a separate CPG for each record type—we do not attempt to combine fields from different records. As previously mentioned, we use the basic block profiling that exists in GCC to estimate the close proximity relation of the field pairs. This is an inter-procedural optimization, therefore the CPG summarizes the close proximity relations across all the functions. The building of the CPG is done function by function. First, we traverse the CFG of a function to build a field reference graph (FRG). Then, we traverse the FRG to calculate the close proximity relation for pairs of field accesses, and finally we summarize this information in the CPG. In the rest of this section we describe the process of building the FRG and traversing it to calculate the close proximity relations. Then we describe our algorithms and heuristics for traversing the CPG and determining field reordering and structure splitting directives.

3.2.1 Building the Field Reference Graph (FRG) from a profiled CFG

We start by traversing the basic blocks of the given function, detecting accesses to fields and

recording them in a chain of accessed fields for each basic block. The fields are chained in the order they appear in the basic block. A chain element represents an access to a field with distance to the next access. The distance is the amount of memory, in bytes, accessed in between the pair of field accesses. A basic block that does not contain accesses to fields will have one dummy element with distance equal to the number of bytes accessed in the block. At the end of the chain we put the distance to the end of the basic block. If memory is accessed at the beginning of the basic block (before the first field access) it is represented as a dummy chain element at the beginning of the chain. After computing the access lists for all the basic blocks of the function, we go over the basic blocks and connect each block's chain to its successors' chains. At the end of this process we have the FRG (Figure 6(b)). Each node in the FRG represents an access to a field. A directed arc between two accesses (f_i, f_j) means that the access f_i is followed by the access f_j. Each arc has a count and a distance. The count is the number of times the sequence (f_i, f_j) occurred, and the distance is the amount of memory accessed in between the access pair.

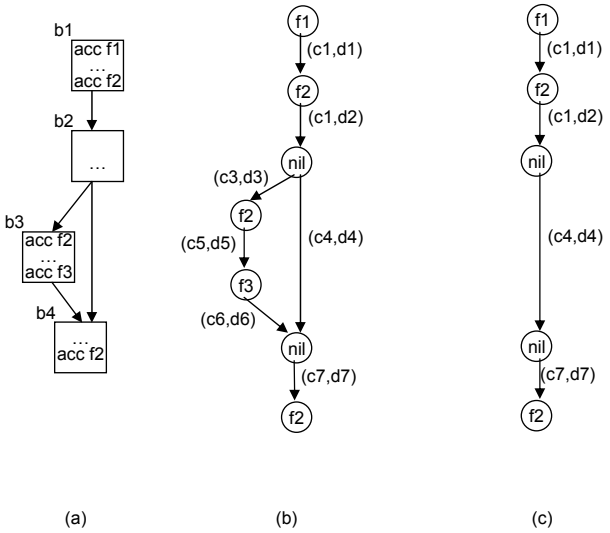


Figure 6: (a) Example Control Flow Graph with accesses to fields. (b) The resulting Field Reference Graph. (c) An example path in the Field Reference Graph.

3.2.2 Collapsing the Field Reference Graph (FRG) and generating the Close Proximity Graph (CPG)

The intent of the FRG is to represent field access sequences, for a more convenient traversal to generate the CPG. A path in the FRG (Figure 6(c)) represents a possible trace of accesses to fields in the instrumented program. We analyze each path and consider only pairs of accesses that are within a predefined distance window. A close proximity relation is added for the accessed pair of fields. The CP relation has two parameters: distance and count. The distance is the summary of the distances along the path from one access to the other. The count is the minimum edge count along that path. For example in Figure 6, the CP for the path from node 1 to node 4 is: (count, distance) = (100, 7). Notice that we take the minimum because the only thing that we are sure about is lower bound on the execution count. The CP relation is added as an edge between the nodes that represent the fields pair. When we have two or more CP relations between a pair of fields we accumulate the counts and average the distances, weighted by the counts. To build the CPG we would like to traverse all the possible paths in the FRG and calculate the CP relation for each pair of fields on each path. The problem with this approach is that in many real programs it has exponential complexity and is therefore not practical. Instead we implemented what we call the collapsing algorithm. The main goal from this algorithm is to get a less complex FRG. Since the complexity of traversing all the paths on the FRG is exponential with the number of split points we want to minimize the split points number.

In Figure 7 we can see that the number of split points was reduced from two to zero. The algorithm we use (presented in Figure 9) has linear complexity (in the number of FRG edges) and it doesn't cause loss of information beyond what the basic block profiling provides. For example in Figure 7, the distance and count from node 1 to nodes 2, 3, and 4 will be the same for the paths that start from nodes 5 and 6. This is true because the distance and count on the edges is the same for all the paths starting from node 1. The collapsing algorithm traverses the FRG nodes looking for candidate nodes—a node is a candidate for collapsing if it has a single predecessor. Once found we call the collapse successor procedure (Figure 8) to update the CPG and record the list of reachable accesses (collapsed successors) in the predecessor node. Then the arc and node are deleted (see Figure 9 for the complete collapsing algorithm). Notice that a loop in the CFG implies a loop in the FRG. Thus when we detect a loop in the FRG we add CP relations for all the possible pairs of the fields accessed within the loop. Another way to look at this is as if the field accesses are ordered in a circle and we go over the circle with a window and add CP relation for pairs that are within the window.

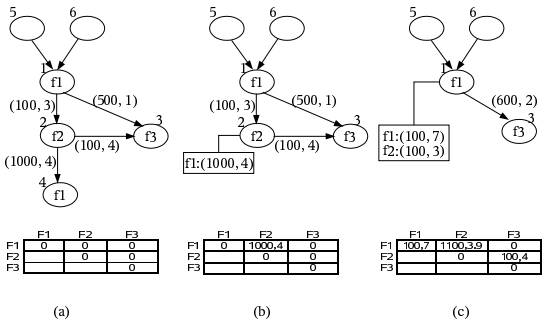


Figure 7: (a) Example FRG and CPG matrix.(b) The FRG and CPG matrix after collapsing node 4 into node 2.(c) The FRG and CPG matrix after collapsing node 2 into node 1.

3.2.3 Algorithms and heurisitc to decide on field reordering

The goal of field reordering is to make fields that are likely to be accessed near each other during the program execution be laid out close together in memory. We use the algorithm described in Figure 10 to determine our field reordering recommendations. This algorithm is taken from [5]. It defines a weight for a given order of a group of fields; this is called the Weighed Close Proximity (WCP). The intent from the WCP is to estimate the expediency—in terms of reducing cache misses—of a given

```
collapse ARC→dest:
foreach collapsed successor of ARC; c_arc
  Update CPG (ARC→src, c_arc→dest),
    with (c_arc→count, c_arc→distance)
  Update CPG (ARC→src, ARC→dest),
    with (ARC→count, ARC→distance)
add ARC→dest to ARC collapsed successors
foreach successor of ARC→dest, SUCC
 foreach collapsed successor of SUCC
 C_ARC, do:
   C_ARC→count
     = MIN (C_ARC→count, ARC→count);
   C_ARC→distance += ARC→distance;
   Update CPG (ARC→src, C_ARC→dest),
   with (C_ARC→count, C_ARC→distance)
   add C_ARC→dest to ARC collapsed successors
 endforeach
 Make SUCC source be ARC→src, with
 minimum (ARC→count, SUCC→count), and
 distance += ARC→distance
endforeach
```

Figure 8: The collapse-successor procedure

```
Collapse (root)
if (root was visited) return false
mark root as visited
change = true
while change is true do:
  change = false
  foreach successor, SUCC, of root
   if (SUCC is a loop arc)
     Update CPG with accesses within SUCC loop
     change = true
   else if (SUCC can be collapsed)
     collapse_successor (SUCC);
     change = true
   endif
   foreach successor, SUCC, of root
     change |= Collapse (SUCC→dest)
endwhile
```

Figure 9: Collapsing algorithm pseudo code

```
F in G | maximum potential WCP
ORDER= [F]; G −= {F}
while (G is not empty)
 foreach(field F in G)
  L= wcp([F] | ORDER); R= wcp(ORDER | [F])
  if (L > max)
   max= L; CHANGE= true; NEXT= F; SIDE= left
  if (R > max)
   max= R; CHANGE= true; NEXT= F; SIDE= right
 endforeach
 if (not CHANGE)
  F in G | maximum potential WCP
  ORDER= [F] | ORDER;
 else
  G −= {NEXT}
  if (SIDE is left) ORDER= [NEXT] | ORDER
  else ORDER= ORDER | [NEXT]
 endif
 G −= {F}; CHANGE= false
```

Figure 10: Field reordering algorithm

ordering. The field reordering algorithm aims to find an ordering that yields maximum WCP. One way to do this is to calculate the WCP of every possible ordering and find the maximum. The problem with this is the complexity of such an algorithm—O(|group of fields|!). Instead we use the heuristic algorithm given in [5]. The main idea is that at any point in the reordering algorithm we choose the field that makes the maximum contribution to the WCP.

3.2.4 Heuristical Algorithm to decide on structure splitting

The goal of this algorithm is to cluster the record in such away that fields that belong to the same sub-record (cluster) are connected with CP relations, which are within a certain threshold. The CP relation threshold is a predefined ratio of the highest CP relation in the sub-record. This ratio is a parameter that we use to tune the splitting algorithm—we call it the `COLD_RATIO`. The dynamic threshold that we use in this algorithm is based on the observation that in many cases the amount of repetition in the field access patterns differ by orders of magnitude. We want to prevent us from ignoring many access patterns becuase there is one that has a very high repition amount. Even access patterns with repetition amounts less than the highest can affect performance. Besides the `COLD_RATIO` there are three additional parameters that we use in this algorithm. The first one is `MAX_SIZE`. We use this parameter to limit the size (in bytes) of each sub-record. This parameter is determined according to the cache line size: If we want the splitting to be efficient we must keep the groups smaller than half of the cache line—we want at least two instances to get into the same cache line. The second parameter is the `DISTANCE_THRESHOLD`. We use this when calculating the CP relation between two fields. When the distance in a CP relation is more than this threshold we ignore the relation (i.e., its value is set to zero). The

third parameter is the SIZE_PENALTY, which is also used in calculating the CP relation. This is used to help give preference to small groups of fields; when we increase the group size we decrease the potential benefit from the splitting because fewer instances will fit in the same cache line. This algorithm is described in Figures 11 and 12.

```
CP(f1, f2) :
cp = CPG [f1,f2]
if (cp→distance > DISTANCE_THRESHOLD)
  return 0;
else if (size(f1) + size (f2) > MAX_SIZE)
  return 0;
endif
max = MAX (size (f1), size (f2))
min = MIN (size (f1), size (f2))
ratio = max / min * SIZE_PENALTY
return (cp→count / ratio)
```

Figure 11: Calculating the CP relation for the Structure splitting algorithm.

3.2.5 Considerations in creating the sub-reord heirarchy

To apply the splitting (clustering) of the record to sub-records, we must make sure that we can get to all the sub-records from a single "root" record—except in the case of pure peeling. Therefore pointers are added according to the heirarchy described in Section 3.3 (see the example in Figure 3). In many cases those additional pointers are brought into the cache when they aren't likely to be accessed in the near future. They also increase the sub-record size, which means fewer instances fit into the same cache lines. Moreover there is the additional overhead of pointer dereferencing when accessing fields of sub-records other than the root of the heirarchy. To minimize the overhead we

```
REMAIN = G; i=1
while (REMAIN is not empty) do
  find (f1, f2) in REMAIN with maximum CP
  CP_THRESHOLD = CP(f1,f2)/(COLD_RATION^N)
  Gi = {f1, f2}
  REMAIN = REMAIN − {f1, f2}
  while (REMAIN is not empty) do
    foreach (fj in REMAIN)
      F_CPj = average {CP(f, fi)|fi in Gi}
      MAX_F_CP
        = MAX{F_CPj|size({fj}+Gi)<MAX_SIZE}
    endforeach
    if (MAX_F_CP ≥ CP_THRESHOLD
      Gi = Gi + {MAX_F}
      REMAIN = REMAIN − {MAX_F}
    else break while
  endwhile
  i = i + 1
endwhile
```

Figure 12: Structure splitting algorithm.

try to organize the heirarchy based on the field access patterns. We distinguish between two kinds of accesses to fields. The first is the general pointer access in which we have a pointer to the record and we want to access its fields. The second is when we access the record using an index into an array of records. In the latter case we can prevent the overhead of pointer dereferencing (pure peeling is when all the accesses to the fields are of the second type). We choose the top of the heirarchy to be the sub-record whose fields have the highest number of accesses of the first type, and the leaves (lowest level of the heirarchy) to sub-records that have the highest number of accesses of the second type.

3.3 Stage 3: Applying the Transformations

Once we have determined which records (if any) it is safe and legal for us to rearrange, and

how we want to rearrange them, we still face the non-trivial task of changing the record definitions and updating the program to use the new data layouts. The following sections describe our implementations of these transformations in GCC.

3.3.1 Field Reordering

Field reordering is the simplest and safest transformation to apply. It does not require changing calls to memory allocation functions, adding new variables, or even updating field access sites. It can be applied safely in cases where variables of types in which we are interested are passed to external library functions (unlike splitting or peeling) because the size of the struct, the number of fields, and the names and types of the fields all remain the same.

To implement field reordering, the first thing we do is to create a new `RECORD_TYPE` which has the same fields as the original record, but in the new order. Then using the cgraph structures we find every global and local variable or parameter whose type was based on the original record, and we replace its type with an equivalent new type based on the newly created record.

3.3.2 Structure Splitting

Implementing structure splitting occurs in five steps:

1. Building the new record definitions.
2. Updating the field mapping information.
3. Updating the program variables.
4. Updating the individual field accesses.
5. Fixing up calls to *malloc*.

Each of these steps is discussed below. These are all performed on the compiler's internal representation of the program.

Building the new record definitions. The first step in structure splitting is the creation of the new record definitions. Stage 2 passes in a tree structure indicating how the original record should be split. There is one node in the tree for each new record to be created. Each tree node contains a list of the fields to be put in the corresponding new record. Each tree node also has up to two outgoing edges, *sibling* and *children*. Sibling nodes are all at an equivalent depth in the new record hierarchy, i.e., all nodes found by traversing only *sibling* edges should be pointed to by the same parent record. Figure 13 shows an example of a tree passed in by Stage 2. This tree would result in the split shown in Figures 2 and 3.

The tree is traversed in postorder, with each node visited being given a unique number representing its order in the traversal. This number is then used to create the name for the new record as well as the name of the field that will point to the new record. This is shown in Figure 14, which is a partial reproduction of Figure 3.

After we have created all the new records, we go back over all of them and check the type of each field to see if any of the fields had a type based on the original record (e.g. a "next" field). If so, we replace the type of such fields with equivalent types based on the root of the new data record hierarchy. It is necessary to wait until all the new records have been created before we do this, as the root record is always created last (postorder traversal of the tree).

Updating the field mapping information. We will need, later on, to know exactly where each

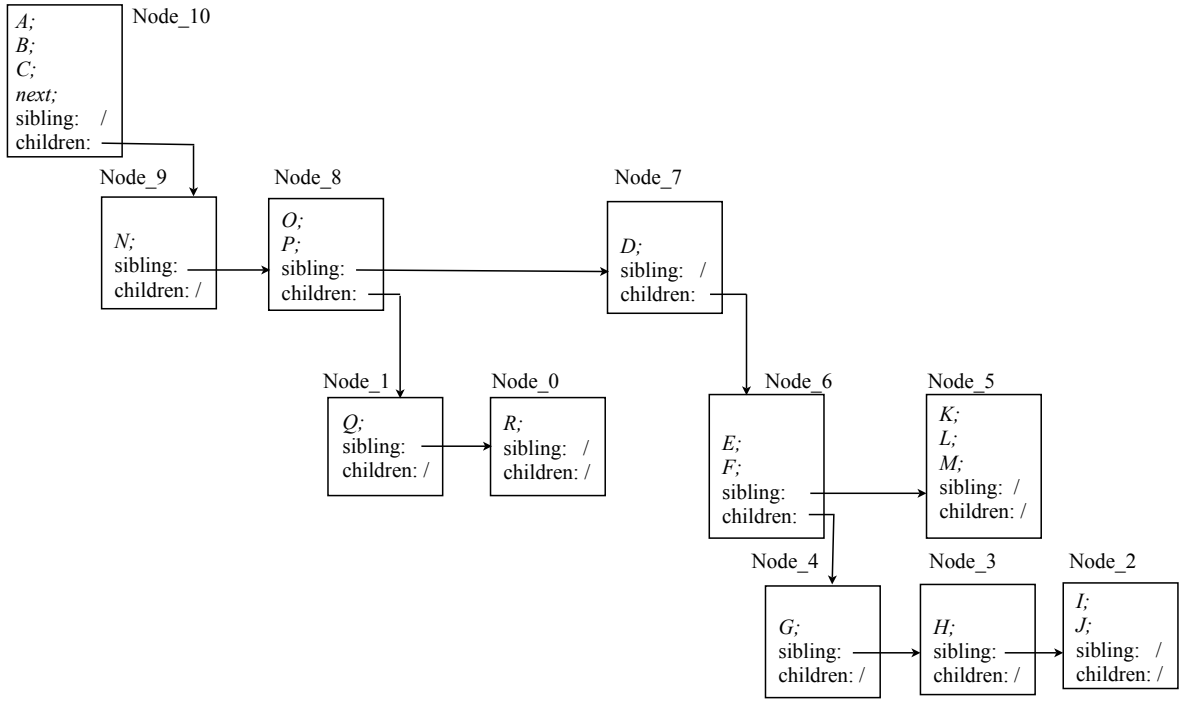


Figure 13: Example of tree structure passed in by Stage 2.

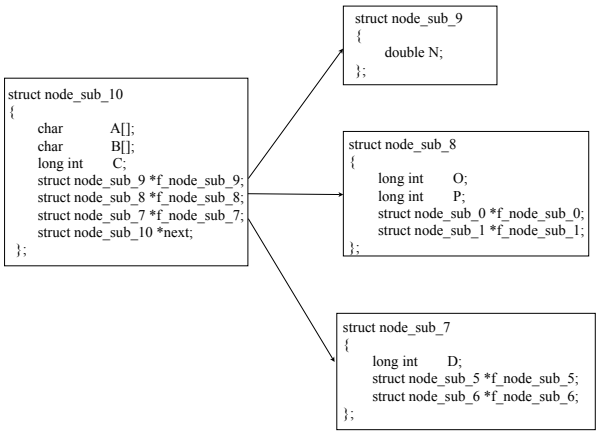


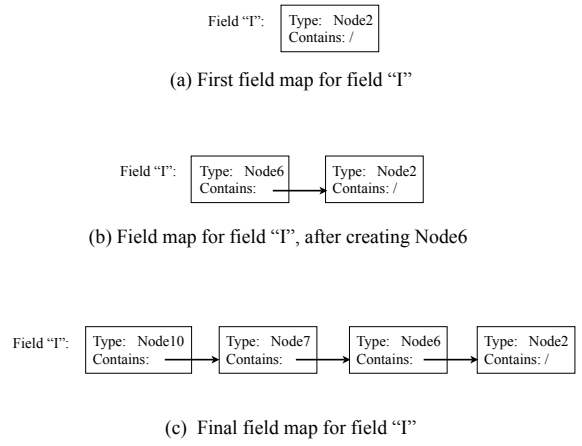
Figure 14: New struct and field names.

field fits into the new record hierarchy, in order to correctly update field accesses within the program. Therefore as we create the new record hierarchy, we update and maintain a set of field mappings.

Each field in the original record has a field map associated with it. The field map is a bi-directional linked list, where each node in the list contains a type declaration, a forward pointer (*contains*) and a backward pointer (*containing_type*).

Whenever a new record is created all the fields that are directly within that record get a new field map consisting of a single node. The type declaration field of the new map points to the newly created record. Next we check the head of each previously existing field map list, to see if its type is pointed to by a field in the new record. If so, we insert a new node at the head of those maps. The new node's type field contains the newly created record. The new node's *contains* pointer now points to the old head of

the field map. This is shown in Figure 15.



(a) First field map for field "I"

(b) Field map for field "I", after creating Node6

(c) Final field map for field "I"

Figure 15: Evolution of field maps as new records are created.

Updating the program variables. After we have created the new record types and updated the field mappings appropriately, we find all the global and local variables and parameters with types based on the original record, and we replace them with new variables with equivalent types based on the new record hierarchy. We also locate every use of the old variables. At each use, we replace the old variable with the corresponding new variable. The new variable declarations are inserted into the appropriate functions.

Updating the individual field accesses. After splitting fields from the original record into several records, we need to find all the places in the program where the fields are accessed and update the access statements to use the new record hierarchy. We take advantage of the fact that all of the field accesses were looked at and recorded in the process of building the CPG. Therefore all we need to do at this stage is go through the list of field accesses already collected, and replace the code that accesses the original field with a new statement or set of statements, based on the new field mapping information we created, to access the field correctly in the new record hierarchy.

```
newNode = (struct student *)
        malloc (sizeof (struct student));
```

(a)

```
(1) T1 = 32;
(2) T2 = malloc (T1);
(3) T3 = (struct student *) T2;
(4) newNode = T3;
```

(b)

Figure 16: Gimple transformation of *malloc* calls.

Fixing up calls to *malloc*. For heap-based dynamic data structures, once we change the size of the nodes we also need to find and fix any calls to memory allocation functions that are based on the original record size. This will usually involve not only changing the size argument for the original function call, but also adding new calls and assignment statements for the newly split off records. Although there are a variety of memory allocation functions, for the sake of brevity we will use *malloc* to represent all memory allocation functions for the rest of this paper.

Before we can correctly update the calls to *malloc*, there are several important pieces of information we need to collect. We need to find every call to *malloc*, and for each call we need to identify the source program variable that is assigned the results of the call and the amount of memory being allocated. Obtaining this information is complicated by the internal representation being in Gimple form [7], which means that the information we need is usually spread across four statements. Figure 16 illustrates this. Figure 16a shows a typical C statement for allocating a new record. Figure 16b shows the Gimple statements generated from the statement in Figure 16a.

We start by finding all assignment statements where the right-hand side is a call to *malloc* (statement (2) in Figure 16b). We do this while the call graph is being built. We organize the calls we find by calling context (the function containing the call), and record for each call the variables corresponding to T_1 and T_2 in Figure 16b.

Next we go through the code for each function that contains calls to malloc, looking for assignments with T_1 on the left or T_2 on the right. This gets us statements (1) and (3) from Figure 16b. Once we have statement (3), we can tell if the call is for a type we are splitting or not. If so, we search forward from statement (3) until we find statement (4), which tells us which source variable we are updating.

Once we have collected all the necessary data about the original *malloc* call, we update the call for the root of our new hierarchy as follows. First we need to calculate the new amount of memory we want to allocate:

(5) numElts = T_1 /
sizeof (originalRecord);
(6) newMallocSize = numElts
* sizeof (newRootRecord);

Then we change statements (2) and (3) as follows:

(2a) T_2 = malloc (newMallocSize);
(3a) T_3 = (struct newRootRecord *) T_2;

The next part is more complicated. In case numElts is not the constant "one" (and we might not be able to tell at compile time) we need to add a loop to iterate over each newly allocated root node, adding the appropriate calls to malloc for the new fields (that point to the other new records) within the new roots. The new loop is roughly equivalent to:

```
for (i = 0; i < numElts; i++) {
  currentRoot = newNode[i];
  updateFieldMallocs(currentRoot);
}
```

where *updateFieldMallocs* for *currentRoot* works as follows. Suppose the original record, *node*, was split into *j* new records, $node_0, node_1, \ldots, node_{j-1}$. To create the new calls to *malloc* for the new fields that point to the new records in the hierarchy, we add the following Gimple statements:

T_0 = sizeof (struct $node_0$);
T_1 = malloc (T_0);
T_2 = (struct $node_0$ *) T_1;
⋮
T_{3j-6} = sizeof (struct $node_{j-1}$);
T_{3j-5} = malloc (T_{3j-6});
T_{3j-4} = (struct $node_{j-1}$ *) T_{3j-5};

The final step is to assign the malloc results (every third statement in the code above) to the appropriate fields. We do this by starting at the root record and doing a depth first traversal of the fields. Any time we find a field whose type is pointer-to-a-new-record, we add a new statement assigning the appropriate allocation results to the field, and then recursively checking the new record it points to.

After we have fixed and added all the necessary calls to *malloc*, the last thing we do is check to see if the original program contained a condition statement that compared the result of the *malloc* call to NULL (a check to verify that the call succeeded). If so, we add appropriate additional verification checks against all the newly inserted malloc calls, and update the CFG appropriately.

3.3.3 Structure Peeling

Structure peeling is a special case of structure splitting, so its implementation involves the same basic steps. Therefore we will discuss only how the implementation of peeling differs from that of splitting.

Building the new record definitions. This step is performed in the same general manner as for splitting, but is simpler, as there is no need to add new fields to point to the newly created records. The result is a set of new, unconnected record definitions.

Updating the field map information. This step is exactly the same as for splitting, but the resulting maps are simpler. Since there is no hierarchy of containing records, the field map lists each contain exactly one node, which points to the new record type that contains the field.

Updating the program variables. This is more complicated than for splitting. Because we are replacing the original record type with a set of record types, we need to replace each variable whose type was based on the original record with a set of variables, one for each new record type. For example, suppose a record, *node*, was peeled into three new records, *node_1*, *node_2* and *node_3*. Then every time we find a declaration of the form

```
struct node foo;
```

we need to replace it with *three* new variable declarations, one for each new record:

```
struct node_0 foo_0;
struct node_1 foo_1;
struct node_2 foo_2;
```

For both splitting and peeling we maintain a list of the "old" variables we find. Each node in the list contains both the tree decl for the original variable and a list of tree decls for the new variables created from it. In the case of splitting, the list of new variables always contains only a single node.

As with splitting, after we create the new variables we locate every use of the old variables that isn't a field access or a call to *malloc* (those are handled separately) and replace them with multiple uses, one for each new variable.

Updating individual field accesses. Since we are dealing with new variables, the first thing we do is look at the field map to find what type the new variable needs to be in order to access the correct field. Then we examine the original field access to determine the original variable used. Finally we find the original variable in our list and look through the new variables generated from it until we find one that has the correct type. Then we substitute the new variable into the field access.

Fixing up calls to *malloc*. Since we are working with arrays, there may not be any calls to *malloc* involved. But if there are any, they will need to be updated.

This works very similarly to splitting, but with two important differences. Rather than the result of calling *malloc* for the new records being assigned to fields within the records, they are assigned to the appropriate newly created variables. Also since we are dealing with arrays, we need to dissect the original call to malloc in order to determine the number of elements, then use that as a multiplier in *all* the new mallocs, similar to the calculations shown in statements (5) and (6) in the previous section.

Updating array element calculations. This is a step unique to peeling; splitting does not require it. Since we are dealing with arrays, we

need to update the array element accesses. In Gimple, "A[i]" has been converted into something like "&A + (sizeof (struct foo) * i)" (see below). Therefore, since we have changed the size of *foo* and replaced *A* with *A_0*, *A_1* and *A_2*, we need to locate the Gimple statements that calculate these array references and update them appropriately. In other words, this:

T_1 = A;
T_2 = sizeof (struct foo);
T_3 = T_2 * i;
T_4 = (struct foo *) T_3;
T_5 = T_1 + T_4;
/* T_5 == A[i] */

becomes this:

T_7 = A_0;
T_8 = sizeof (struct foo_0);
T_9 = T_8 * i;
T_{10} = (struct foo_0 *) T_9;
T_{11} = T_7 + T_{10};
/* T_{11} == A_0[i] */

⟨Repeat for A_1 and A_2⟩

4 Measurements

In order to measure the effectiveness of our optimization we took a series of measurements. We started by turning on the optimization for the SPEC2000 C benchmarks, to see which benchmarks it determined it could transform. The optimization applied structure peeling to structs in *equake* and *art*, and it applied field reordering to structs in *gzip*, *mcf*, and *crafty*. It did not apply full structure splitting to anything because the implementation of full splitting is not complete at the time we are writing this.

4.1 Methodology

We ran our measurements on two systems with different CPUs and cache parameters: An Apple G4 and an Apple G5, both running versions of Mac OS X. Table 1 shows more details about those systems. The measurements we took included running the SPEC benchmarks using the SPEC harness to get a standard SPEC result number (however, not official SPEC runs); and running the benchmarks manually (without the harness) to count the number of hits and misses in the various caches. We measured all of this data first without our optimization, and then with it, in order to determine how our optimization changes the behavior of the programs. The numbers shown here are averages taken over multiple runs of each program, with the appropriate measurements taken during each run. We were careful to alternate which versions of programs were being run, never running the same one twice in a row, in order to avoid having a "warm" cache at the start of a measurement run.

4.2 Results and Analysis

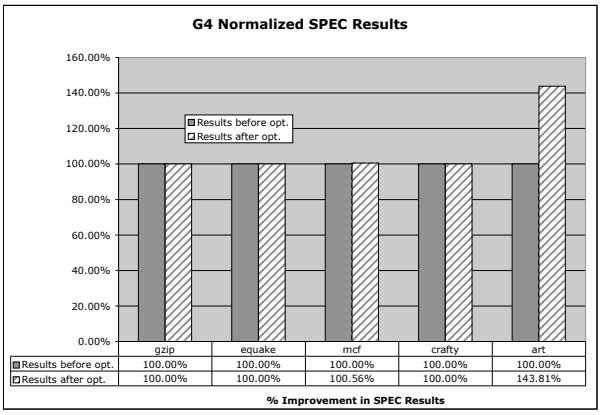


Figure 17: G4 SPEC Performance Results

Figures 17 and 18 show the SPEC performance results on the G4 and the G5. As can be seen, the optimization did not have much impact on

Hardware	OS	CPU Speed	DL1 Cache	L2 Cache	L3 Cache
G4	Mac OS X 10.3.9	800 MHz	32 KB	256 KB	2 MB
G5	Mac OS X 10.3.8	2 GHz	32 KB	512 KB	none

Table 1: Various hardware configurations used for measurements.

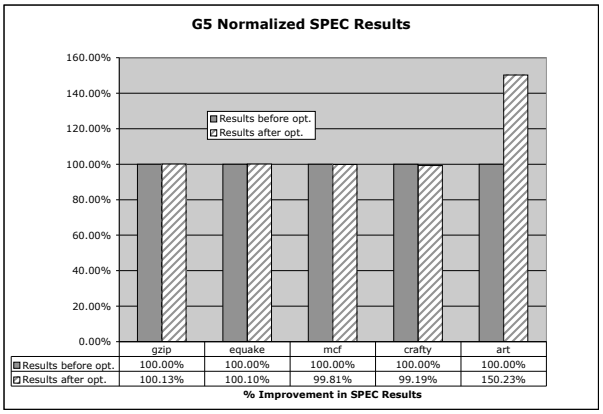


Figure 18: G5 SPEC Performance Results

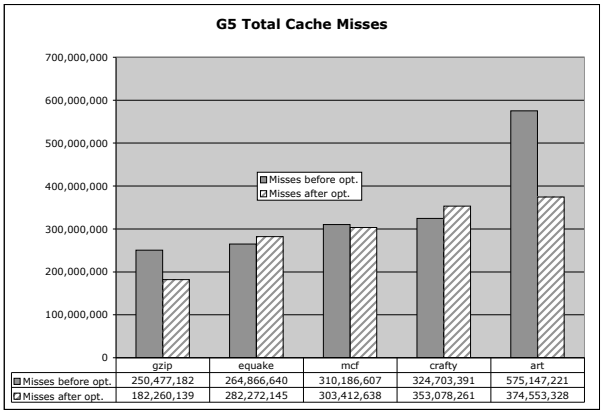


Figure 20: Optimization effect on total cache misses on a G5

the performance of most of the benchmarks. However it dramatically improved the performance of *art* on both architectures (roughly 47% improvement). We then looked at the cache miss data for both architectures. Figures 19, 20, and 21 show the cache miss data for the G5. One thing that is immediately apparent is that *art* is the only benchmark for which the number of L2 misses is significant, compared to the number of L1 misses. It is worth pointing out that the G5 does not have an L3 cache, so it has far less cache than many other architectures. Therefore cache misses on a G5 can cause far more performance problems on a G5 than on other architectures. Given that *art* is the only benchmark where the number of L2 misses far outweighed the number of L1 misses, it is reasonable that *art* was the only benchmark to show much improvement on the G5.

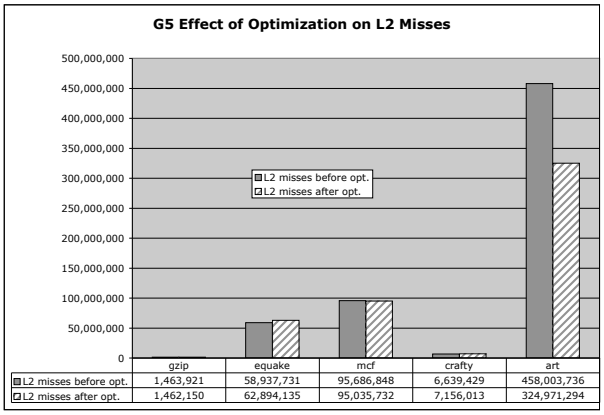
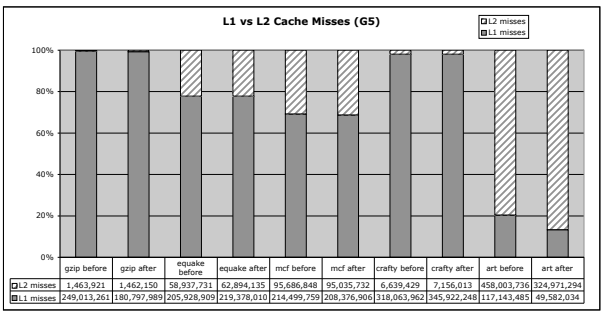


Figure 21: Optimization effect on L2 cache misses on a G5

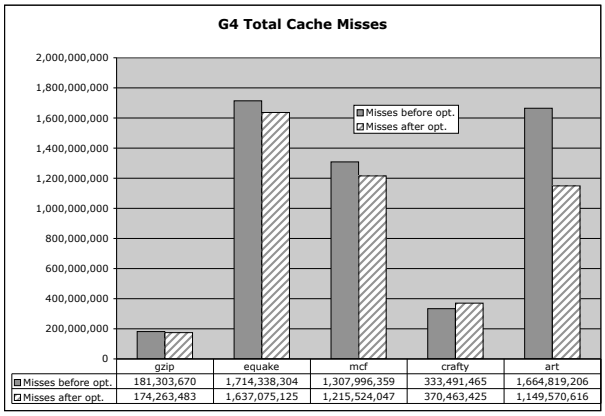
Figures 22, 23, 24, and 25 show the cache miss data for the G4. This data does not so clearly



	gzip before	gzip after	equake before	equake after	mcf before	mcf after	crafty before	crafty after	art before	art after
L2 misses	1,463,921	1,462,150	58,937,731	62,894,135	95,686,848	95,035,732	6,639,429	7,156,013	458,003,736	324,971,294
L1 misses	249,013,261	180,797,989	205,928,909	219,378,010	214,499,759	208,376,906	318,063,962	345,922,248	117,143,485	49,582,034

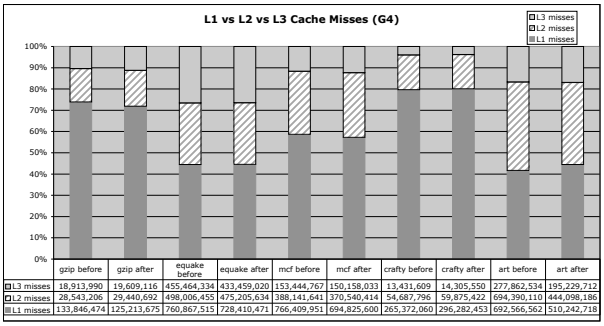
Figure 19: Relative misses in L1 cache versus L2 cache on a G5

match the SPEC performance results and needs a bit more analysis. Looking only at the cache miss data, our optimization appears to reduce the number of misses in all three caches for all the benchmarks except *crafty*. Most of them get just a little better, *art* gets a lot better, and *crafty* gets a little worse. One might expect runtime performance to mirror this, but it didn't: *art* got better, while all the others remained virtually unchanged. We then looked at the L1 cache miss ratios, to see if perhaps the number of hits in the L1 cache was large enough to overshadow the changes in the number of cache misses. Figures 26 and 27 show the L1 hits and misses both with and without the optimization, as well as the optimization effects on the miss ratios. Looking at the miss ratios we see that the optimization causes *gzip* and *equake* to get a little better, *mcf* and *crafty* to get worse, and *art* to get much better. The last thing we looked at were the relative cache misses (unoptimized) between the various caches. Looking at this we see that both *crafty* and *gzip* had very few L3 cache misses: they are nearly an order of magnitude less than their L1 misses, and (for *crafty*) nearly two orders of magnitude less than the L1 hits. This implies that *gzip* and *crafty* almost never miss in the cache, so our optimization would not have any visible effects on them. *Art*, with significant improvements in both its cache misses and its miss ratio, has a matching performance improvement. *Mcf* gets some improvement in its cache misses, but this is offset by getting worse in its miss ratio, so overall we would not expect it to improve much. *Equake* is a bit of a disappointment. It has a lot of misses in the L3 cache, and its number of hits in the L1 cache is roughly proportional to its misses in the L3 cache, so there ought to be lots of room for our optimization to have improved its performance. It seems likely at this stage, that the structure our optimization peeled was not as important to performance as other structures in the benchmark. Although the optimization did improve both its cache misses and its miss ratio, the improvements were relatively small and were overshadowed by the large number of cache accesses the program made.



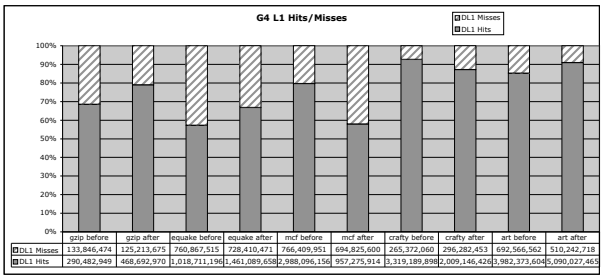
	gzip	equake	mcf	crafty	art
Misses before opt.	181,303,670	1,714,338,304	1,307,996,359	333,491,465	1,664,819,206
Misses after opt.	174,263,483	1,637,075,125	1,215,524,047	370,463,425	1,149,570,616

Figure 23: Optimization effect on total cache misses on a G4



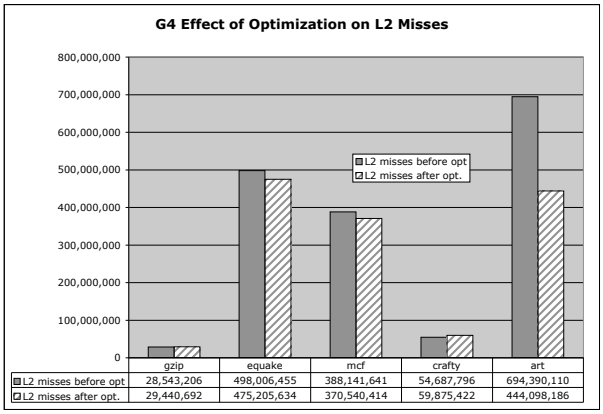
	gzip before	gzip after	equake before	equake after	mcf before	mcf after	crafty before	crafty after	art before	art after
L3 misses	18,913,990	19,609,116	455,464,334	433,459,020	153,444,767	150,158,033	13,431,609	14,305,550	277,862,534	195,229,712
L2 misses	28,543,206	29,440,692	498,006,455	475,205,634	388,141,641	370,540,414	54,687,796	59,875,422	694,390,110	444,098,186
L1 misses	133,846,474	125,213,675	760,867,515	728,410,471	766,409,951	694,825,600	265,372,060	296,282,453	692,566,562	510,242,718

Figure 22: Relative misses in L1 cache versus L2 cache versus L3 cache on a G4



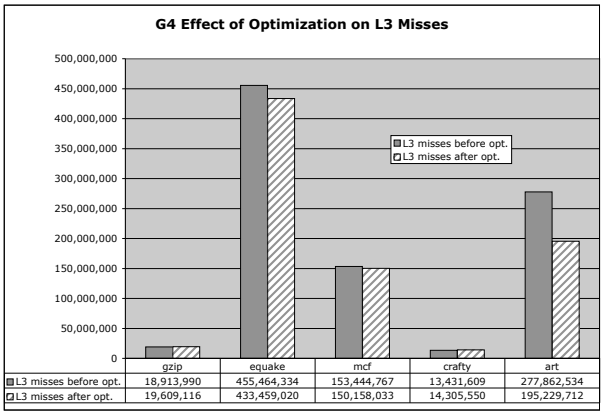
	gzip before	gzip after	equake before	equake after	mcf before	mcf after	crafty before	crafty after	art before	art after
DL1 Misses	133,846,474	125,213,675	760,867,515	728,410,471	766,409,951	694,825,600	265,372,060	296,282,453	692,566,562	510,242,718
DL1 Hits	290,482,949	468,692,970	1,018,711,196	1,461,089,658	2,988,096,156	957,275,914	3,319,189,898	2,009,146,426	3,982,373,604	5,090,027,465

Figure 26: G4 L1 cache relative hits and misses



	gzip	equake	mcf	crafty	art
L2 misses before opt	28,543,206	498,006,455	388,141,641	54,687,796	694,390,110
L2 misses after opt.	29,440,692	475,205,634	370,540,414	59,875,422	444,098,186

Figure 24: Optimization effect on L2 cache misses on a G4



	gzip	equake	mcf	crafty	art
L3 misses before opt.	18,913,990	455,464,334	153,444,767	13,431,609	277,862,534
L3 misses after opt.	19,609,116	433,459,020	150,158,033	14,305,550	195,229,712

Figure 25: Optimization effect on L3 cache misses on a G4

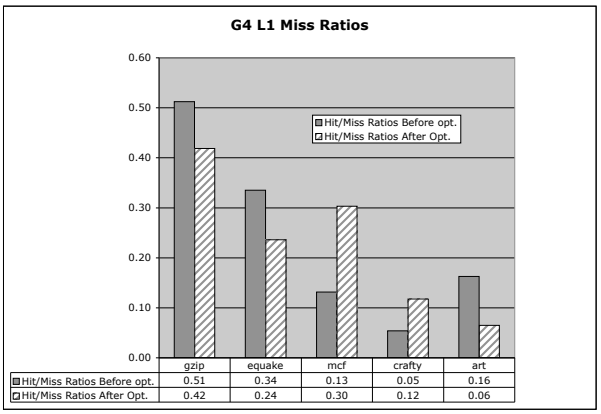


Figure 27: Optimization effect on G4 L1 cache miss ratios

5 Current Status and Future Work

This work is being developed in the struct-reorg-branch of GCC. Currently we have implemented all of Stage 1 and Stage 2, and most of Stage 3. The field reordering and structure peeling optimizations are completely implemented; the full structure splitting optimization is mostly implemented, but the part responsible for deciding on the sub-record hierarchy (described in Section 3.2.5) is not implemented yet. Our hope is to have full structure splitting working by mid-summer in 2005.

Once that is done, there are several other areas we could pursue. It was disappointing to only gain performance in one benchmark. It would be interesting to do further analysis as to why more benchmarks did not benefit as much as might be expected, and perhaps tune our optimization further based on such analyses. It might also be fruitful to look at relaxing some of the safety and legality constraints: perhaps we are being overly conservative in applying our optimization, and if we could overcome this we might see more performance gains. Another possible direction is to influence malloc to place instances of the same subrecords in continuance memory locations. In addition we may discover a need to improve our heuristics as we broaden the range of programs to which we attempt to apply this optimization.

We also earnestly wish to pursue obtaining feedback from the GCC development community with respect to our work, and hope to eventually be able to integrate this optimization into an official release of GCC.

References

[1] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.

[2] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 1999.

[3] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[4] Andre Seznec Dan N. Truong, Francois Bodin. Improving cache behaviour of dynamically allocated data structure. *PACT 98*, 1998.

[5] Mostafa Hagog. Optimization techniques for locality of data structures. Master's thesis, Technion - Israel Institute of Technology, March 2001.

[6] David Patterson John L Hennessy. *Computer architecture a quantitative approach*. Second edition, 1990.

[7] Jason Merrill. Generic and gimple: A new tree representation for entire functions. *GCC Developers Summit*, 2003.