Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements

Gautam Chakrabarti, Fred Chow

PathScale, LLC.
{gautam, fchow}@pathscale.com

Abstract

A common performance problem faced by today's application programs is poor data locality. Real-world applications are often written to traverse data structures in a manner that results in data cache miss overhead. These data structures are often declared as structs in C and classes in C++. Compiler optimizations try to modify the layout of such data structures so that they are accessed in a more cache-friendly manner. These compiler transformations to optimize data layout include structure splitting, structure peeling, and structure field reordering. In this paper, we present the design and implementation of the transformations of structure splitting and structure peeling, in a commercial version of the Open64 compiler. We present details of how these transformations can be achieved in the Open64 compiler, as well as the analyses required to safely and usefully perform the transformation. We present some performance results from the SPEC CPU2000 and the CPU2006 suites of benchmarks to demonstrate the effectiveness of our implementation. Our results show that for some applications these layout optimizations can provide substantial performance improvement.

Keywords data cache misses; data layout; data locality; inter-procedural optimization; profile data; reference pattern; structure layout transformation; whole program analysis

1. Introduction

Many of today's application programs exhibit poor data locality. They often access large data sets, or access them in a manner that may cause cache thrashing. This behavior causes them to suffer from high data cache miss penalties. This situation is worsened by the growing gap between processor and memory speeds, causing the CPU to stay idle more often while waiting for data from memory.

Compilers can play a role in improving data cache usage. One class of compiler optimizations that helps this situation is loop transformation, which attempts to modify the order in which the data are accessed. A second class of optimizations aims to change the layout of the data structures to make the application more cache-friendly. This second class of transformations has program-wide effects because they affect all accesses of the modified data structure. In this category, one approach is to re-layout stack variables, global variables, and heap-allocated objects in a cache-friendly manner. Another approach is to modify the layout of data types based on accesses of fields in the structure.

This paper addresses the latter approach of changing the layout of structure types. Our approach requires the compiler to perform whole-program analysis and determine the access patterns of fields of hot structures. Based on available dynamically generated profile feedback data, or on statically estimated frequency counts, our analysis attempts to detect the data types that are the most beneficial to transform and then determines their desired layout. Subsequent compiler transformation changes the order of data members (or fields) in the structure according to recommendations from the analysis.

This work is implemented in the PathScale version of the Open64 compiler for X86 processors. The main contribution of this work is in development of the framework to perform this structure layout optimization within the Open64 compiler infrastructure. This involves separating out type information summarization into the pre-IPA phase called IPL. To fit the IPA framework in the Open64 compiler, we also developed an algorithm for deciding which structures to transform; this algorithm is different from techniques presented previously.

This paper is organized as follows. In Section 2, we briefly describe the Open64 compiler infrastructure including the IPA (Inter-procedural Analysis and Optimization) compilation framework. Section 3 describes the structure layout transformations called structure splitting, structure peeling, and field reordering. Section 4 details the

analysis and transformation for the structure-layout optimizations. We present our performance results in Section 5. We discuss related research work in Section 6. Finally, we conclude and discuss scope for future work in Section 7.

2. Open64 Compiler Infrastructure

We implemented the structure layout transformation in the Open64 compiler. This transformation changes the layout of structure types, and hence changes the actual definition of such types. As a result, such an optimization needs to affect all usage of such types in the whole program. Hence, we transform such types in the Inter-Procedural Analysis and Optimizations (IPA) phase of the Open64 compiler. This phase is enabled by the -ipa option.

When IPA is enabled, the compiler driver first invokes the language-specific front-end on each user source file. The next phase, called IPL, analyzes its input and stores the results of the analysis in the form of summary data. This data summarizes all the information that would be used by inter-procedural analysis during whole program optimization. At the end of IPL, it outputs intermediate object files that contain the WHIRL intermediate representation as well as summary data.

IPA has an analysis phase and an optimization phase. During the analysis phase, IPA works on the summary data and does not open the WHIRL Intermediate Representation (IR). The summary data can include all information that is needed by IPA in the analysis phase. For example, the summary may include information about all functions and subroutines presented to the compiler. As a result, such data enables IPA to easily and efficiently perform its analysis without needing to open IR files. In the analysis phase, IPA decides the transformations that need to be done during the optimization phase. The optimization phase opens the IR for each subroutine and performs the transformations, before writing out the optimized subroutine. Depending on the size of the application, IPA outputs the transformed routines in a number of output files, on which the back-end is run to generate the assembly output. Hence, the backend compilation of the output files can be done in parallel. Figure 1 has an illustration of the compilation flow during IPA, where "pathcc" invokes the PathScale™ compiler.

Most of the analysis for performing the structure layout optimization is done before IPA by the front-end and IPL. These analyses include determining what data types can be legally transformed, as well as estimating the benefits from transforming a certain data type. IPA's analysis phase aggregates all summary data and determines the data types to be optimized, as well as the new layout of such types. The actual transformation is performed in IPA's optimization phase. It includes updating the data structure layout in the symbol tables, as well as modifying the WHIRL IR.

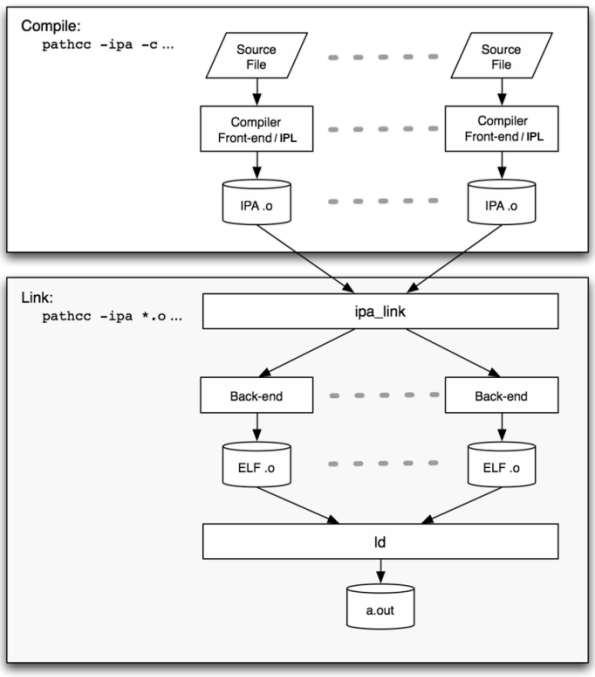

Figure 1: IPA compilation flow diagram

3. Types of data-layout optimizations

The layout of structures can be modified in several ways, as presented below. Similar forms of structure splitting and structure peeling have previously been proposed by Hagog et al [1] and Hundt et al [2].

```
struct struct_A
{
 double d1;
 double d2;
 int i;
 float f;
 long long l;
 char c;
 struct struct_A * next;
};
```

(a)

```
struct new_struct_A
{
 double d1;
 int i;
 long long l;
 struct new_struct_A * next;
 struct cold_sub_struct_A * p;
};
```

(b)

```
struct cold_sub_struct_A
{
 double d2;
 float f;
 char c;
};
```

(c)

Figure 2: An example structure definition (a), hot sub-structure after splitting (b), cold sub-structure after splitting (c)

- **Structure splitting:** This optimization splits a structure into multiple parts, and inserts pointers in the parent structure to access all the newly created sub-structures through these pointers. In the general case, each of these sub-structures can be recursively split, as shown in [1]. Figure 2(a) shows an example structure. Let us assume that the first (*d1*), third (*i*), fifth (*l*), and the last field (*next*) are hot fields. After splitting, the hot fields are grouped together into a structure shown in Figure 2(b). The cold fields are separated out into a different sub-structure (Figure 2(c)). The structure with the hot fields is considered the root of the hierarchy, and contains a link pointer to point to the sub-structure with the cold fields. This transformation introduces a pointer indirection (through *p* in the example) and hence, increases the overhead to access the fields in the child structure. As a result, the cold fields are laid out in the child structure, so that no overhead is added in accessing the hot fields of the structure. The transformation also increases the size of the hot structure by a pointer size.
- **Structure peeling:** This is a type of structure splitting that does not require the insertion of the extra pointers in the parent structure. The name *structure peeling* has been introduced in [1]. Figure 3(a) contains the structure definition from Figure 2(a), except it does not contain the field *next*. As a result, the structure example is not a recursive data structure, in contrast to the previous example. This fact indicates to the compiler that it can perform structure peeling instead of structure splitting. Thus, it can prevent inserting an extra pointer and also the overhead of indirection through that pointer. Assuming the first (*d1*), third (*i*), and fifth (*l*) fields of the structure are hot, the compiler peels the structure to generate a hot structure (Figure 3(b)) and a cold structure (Figure 3(c)). It is to be noted that the compiler can still peel a structure that is an indirectly recursive data structure through multiple structures.
- **Field reordering:** Based on the hotness of fields in a structure, this transformation modifies the order in which the fields in the structure reside. This optimization is done by IPA in the Open64 compiler, but is not the subject of this paper.

We describe the implementation of structure splitting and structure peeling. We will provide the performance impact of this transformation in Section 5.

```
struct struct_A
{
 double d1;
 double d2;
 int i;
 float f;
 long long l;
 char c;
};
```

(a)

```
struct hot_struct_A
{
 double d1;
 int i;
 long long l;
};
```

(b)

```
struct cold_struct_A
{
 double d2;
 float f;
 char c;
};
```

(c)

Figure 3: An example structure definition (a), hot sub-structure after peeling (b), cold sub-structure after peeling (c)

4. Performing structure-layout transformations

It is not always safe to change structure layout in programs. This is because a programmer often uses knowledge of data layout in a structure definition for writing an application. The compiler has to detect such situations and prevent optimizing such data types to ensure correct execution of the application. We present this analysis in Section 4.1.

The compiler also needs to analyze data structures in a program to determine the structures that are most profitable to transform. It also needs to determine the desired layout of the structures that should be transformed. This analysis is discussed in Section 4.2.

Section 4.3 describes our implementation in the Open64 compiler in detail.

4.1. Legality Analysis

The following are the main legality checks that our implementation in the Open64 compiler does. For any structure type *T*:

Type cast: If a cast to a *pointer-to-T* is found, it indicates unsafe usage of *T*, and so *T* is invalidated. Similarly, a cast from a *pointer-to-T* also causes *T* to be invalidated.

There is an exception to this rule. During dynamic memory allocation in C/C++, the library routines return (void *). As a result, when memory is allocated to hold objects of type *T*, it will involve a type cast from type (void *) to *pointer-to-T*. In WHIRL IR, when the return value is loaded from such a routine, it is given the final type as seen after the type cast. As a result, such type casts are hidden and, hence, does not prevent the optimization of the type.

This type casting restriction is imposed in IPL. The IPL phase processes each function (also called PU, short for Program Unit) by traversing its WHIRL and summarizing the data. During this step, IPL also scans for type casting that might prevent a structure type (called TY in Open64) from being split or peeled. It marks such structure TYs appropriately, so that the information can be used by IPA's analysis phase while determining transformable types. IPL's flagging of such TYs is discussed in more detail in Section 4.3.1.

Address of a field is taken: If the compiler detects the address of a field of a structure to have been taken, then it implies the application may have address arithmetic on the data fields. As a result, the type needs to be invalidated.

Escaped types: A type cannot be split if it escapes to code outside the scope of analysis. This is because the compiler cannot know whether that code outside its analysis scope accesses the type in any unsafe manner.

Parameter types: If a type is passed to a routine as an argument or is returned from a routine, we do not peel that type. The reason is peeling such a type would require introducing new parameters to the function, and hence changing the function prototype. Currently we prevent such types from being optimized due to implementation limitations.

Whole program analysis: Code that is written as part of a program can freely access the internal data structures without their passing as parameters to a function. As a result IPA needs to ensure it is given the entire application code, so that it can perform legality checks on the whole program. The IPA linker ("ipa_link") can impose this legality check by ensuring that all object files presented to it during whole-program compilation are IPA object files. If the application passes this check, it means IPA is able to analyze the entire application code. IPA can exclude some system object files and libraries that may be passed to it from this restriction, because such files cannot access the data structures of the user application.

Alignment restrictions: If the user explicitly enforces alignment or packing restrictions on a structure type, then that type is invalidated.

4.2. Profitability Analysis

In order to estimate the application performance benefit that transforming a data structure might bring, we mainly perform two types of analyses in the Open64 compiler: *hotness* analysis and *affinity* analysis. The terms *affinity* and *affinity group* (used later) have been used in [2]. We estimate the number of accesses to structure fields in a program. A field that is accessed more frequently than other fields is hotter. Currently, we only consider field accesses inside loops to determine their hotness. In a program, we also analyze accesses to fields in a structure to determine "closeness" between the fields. We call two structure fields to have high affinity when the fields are accessed close together in time. Fields that are consistently accessed close to each other are considered to have higher affinity than others. The affinity is computed at the loop-level which means that fields accessed inside the same loop are considered affine to each other.

4.3. Analysis and Transformation Implementation Details

In the Open64 compiler, the implementation of the structure layout transformations is divided into the three phases: type information summarization, analysis, transformation.

4.3.1. Type information summarization

For each source file, we determine structure types that fail any of the legality checks and summarize information about structure field accesses. The Open64 phase before IPA, called IPL, is run over all the input files. This phase traverses the PUs and marks TYs that violate any of the legality checks. We added a TY flag TY_NO_SPLIT to mark types that cannot be split, which applies only to structure TYs. This flag, however, cannot be set on a TY by a phase before IPA. This is because when IPL traverses the WHIRL and invalidates types, a certain type may be marked invalid while processing one source file. In another file, that type may not have any usage that fails the legality restrictions. As a result the type definition present in that file will not be invalidated. In such a scenario, a certain type *T* may have the flag TY_NO_SPLIT set in one file, but not in another. When IPA merges the symbol tables from all its input files, it compares the symbols and types byte-by-byte to determine if any of them are identical and can be merged. As a result, it will fail to merge type *T* from one file with type *T* from another file if they have different flags set.

We addressed this issue by introducing a summary data structure in IPL, called SUMMARY_TY_INFO. IPL invalidates a TY by recording the information in the summary for that TY. IPA reads in the array of SUMMARY_TY_INFO and updates the merged TY table based on the flags in this summary data. This summary data structure may also be used in future for other information that IPL needs to convey about a type to IPA.

While traversing a PU, IPL also summarizes information about structure field accesses. This access information is used by IPA to compute hotness of and affinity between fields. We introduced a summary data structure in IPL, called SUMMARY_LOOP. IPL maintains a SUMMARY_LOOP data structure for each loop (DO_LOOP, WHILE_DO, and DO_WHILE in WHIRL). For each loop, it tracks up to *N* structure TYs, where *N* is a tuning parameter. For each such TY, the summary data contains a bit-vector to keep track of which fields are accessed inside the loop. For a loop, SUMMARY_LOOP only tracks field accesses that are contained immediately inside the loop, and does not count accesses that may be present in a nested loop. From the example in Figure 4, the summary data for *Loop 1* only counts field accesses in *statement 1*, *statement 2*, and *statement 4* in *Loop 1*. The heuristic used here is that fields accessed inside a nested loop (fields 3 and 4 in the figure) are considered affine to each other, but are not considered "close" to fields accessed in a parent loop (fields 1 and 2 in the figure).

```
for ( i = 0; i < N; i++ )       // Loop 1
{ // Field accesses below are to struct S
  // Loop 1 statement 1 accesses field 1
  // Loop 1 statement 2 accesses field 2
  for ( j = 0; j < M; j++ ) // Loop 2,
                            // Loop 1 statement 3
  {
    …
    for ( k = 0; k < L; k++ )  // Loop 3
    {
      // Loop 3 accesses fields 3 and 4
    }
  }
  // Loop 1 statement 4
}
```

Figure 4: A triply-nested loop accessing fields of struct S

Each SUMMARY_LOOP also has a field to store the estimated execution count of the statements immediately contained inside the loop. When runtime feedback is enabled, this invocation count is the information obtained from the profile data. In the Open64 compiler, a feedback run is obtained by using the compiler options *–fb-create fbdata* and *–fb-opt fbdata* in the two phases respectively. Without profile data, we developed a framework that computes static profiles using heuristics [3] during compile-time. The execution count for the SUMMARY_LOOP is obtained by employing these compile-time heuristics. For example, without profile feedback, loops are assumed to execute 8 times. As this execution count is assigned in IPL, the estimate is local to the current PU, and is independent of how many times the PU itself is called. Hence, IPA needs to fix up this estimated execution count, which we discuss below.

4.3.2. IPA Analysis

After building the IPA call graph early in the analysis phase, we added a pass in which IPA traverses the call graph to update the statically estimated execution count of the PUs. In a top-down traversal over the call graph, IPA scales up the

invocation count of a PU based on the invocation count of its callers, and the number of times the PU is called from each invocation of the callers. This pass is similar to the approach mentioned in [3]. It then uses this scaling factor to also update the statically estimated loop frequencies obtained from the SUMMARY_LOOP data structure.

In the analysis phase, IPA scans through the TY table to determine structure TYs that are candidates for the structure-layout optimization. The decision of whether to split or peel a structure and the choice of its new layout depends on a number of factors including the type definition of the structure, and on the objects of that type that the program has. As seen from Figure 2, if a structure type is a recursive data structure that contains a pointer to itself, then the type may only be split. In such a scenario, the compiler does not have the lower-overhead option of peeling it.

	F_4	F_3	F_2	F_1	BV
L_1		22		22	0101
L_2			14		0010
L_3		12		12	0101
L_4	8	8			1100
L_5		6		6	0101

Table 1: Count of field accesses in loops L_1 to L_5

	F_4	F_3	F_2	F_1
AG_1		40		40
AG_2			14	
AG_3	8	8		

Table 2: Aggregated field access pattern for field affinity computation

In order to determine the new layout of a structure type *T* that has not been invalidated, we analyze the structure field accesses from the loop summary data of the hottest *P* PUs, where *P* is a configurable parameter. For structure type *T*, we maintain a list of fields that are accessed in each of the hottest loops considered. Let us consider the example shown in Table 1 for a structure type *T* that is a candidate for structure peeling. Let us assume that *T* has 4 fields F_1-F_4. We are analyzing the accesses of these fields in the 5 hottest loops numbered L_1-L_5 shown in order of decreasing hotness in the rows of the table. The right-most column marked BV is a bit-vector that gives the field accesses of the structure for the corresponding loop. We list the fields F_1-F_4 in reverse order in the columns of the table to make it easier to match them with the corresponding bit-vector. The numbers in the columns F_4-F_1 indicate the access count for the corresponding field. Fields that we consider "close" to each other belong to the same affinity group. In order to compute the affinity between the fields, we begin scanning the loops starting with the hottest loop. For two loops that have the same access pattern for the fields of the structure, we aggregate the field access counts and merge them into a single affinity group. Two loops L_i and L_j have same access pattern for a structure type *T* when the fields of *T* accessed in L_i are exactly the same as those accessed in L_j. From our example in Table 1, we find that loops L_1, L_3, and L_5 have the same bit-vector signifying the same access pattern. We aggregate them and form affinity group AG_1 (Table 2). Hence, the analysis of the loops in Table 1 results in the affinity groups in Table 2. As the affinity groups AG_2 and AG_3 have a very low hotness factor compared to AG_1, and have a hotness factor less than the hotness threshold T_h, AG_1 takes priority and we aggregate all the remaining fields into another affinity group. T_h is a tunable parameter. If there are multiple non-disjoint hot affinity groups (example: if both AG_1 and AG_3 were hot), we scan the affinity groups in decreasing order of hotness, and form affinity groups using fields that have not already been assigned to a group. In general, if there are multiple affinity groups that have hot fields, then the compiler will peel the original structure *T* to create multiple sub-structures. Each sub-structure will correspond to one affinity group. However, if an object of type *pointer-to-T* exists as a field in another structure *S*, then we peel *T* into at most 2 sub-structures. The reason is the transformation necessitates the compiler to insert fields in *S*. If *T* is peeled into N sub-structures, then the compiler must insert (N – 1) new fields in structure *S*, thus increasing the size of *S* by (N – 1) times the pointer size. To reduce this overhead to a minimum, we enforce N=2. In this case, the compiler inserts one new field in structure *S*.

As a special case, if all or most of the hottest loops access only one field F_i in *T*, then F_i becomes the only member of the hottest affinity group. If all the other fields of *T* are relatively cold, then they all form another single affinity group. In this scenario, the compiler will peel the structure into two sub-structures T_1 and T_2 such that T_1 contains the lone field F_i, and T_2 contains the remaining fields. As T_1 is a structure containing only one field, it can be simplified into the type of F_i. Figure 5(a) shows an example of such a structure. Figure 5(b) shows the result of peeling out the lone hot field of type "double". The object of type *pointer-to-T*, *p*, is replicated to give *p_1* and *p_2*. The object *p_1* is a

```
struct T
{
  double hot_field;
  int cold_field1;
  float cold_field2;
};
struct T * p;
```

(a)

```
struct T2
{
  int cold_field1;
  float cold_field2;
};
double * p_1;
struct T2 * p_2;
```

(b)

Figure 5: Original structure example (a) and result of peeling out the first field in (b)

pointer to the hot field, while object *p_2* is a pointer to the newly created structure type.

If we are analyzing structure type *T* that is a candidate for structure splitting, we analyze the field accesses in the hottest loops from Table 1 and divide them into two categories: hot fields and cold fields. As shown in the example in Figure 2, we create two structures, one of which is made the parent or root structure. The fields that are hotter than a configurable hotness threshold T_h constitute the parent structure. The child structure is formed from the remaining cold fields. A new field pointing to the cold structure is added to the end of the root structure. As a result, this new field increases the size of the parent structure. To be beneficial, the total size of fields in the cold structure should be greater than a pointer size. It also requires one extra level of pointer traversal to access the cold fields. In order to minimize these overheads, we keep the layout simple and form a hot structure and a cold structure. We feel that in general the benefits of splitting a structure into many sub-structures based on affinity and hotness criteria are offset by the overhead introduced by the transformation.

4.3.3. Transformation

Once IPA's analysis phase determines the structure-layout transformation to be done on a structure type, IPA's optimization phase is ready to perform the actual transformation. This transformation process involves updating the WHIRL symbol tables, as well as the WHIRL IR. The following are the main steps:

Adding new type definitions: IPA traverses the layout suggested by the analysis phase and forms new structure types based on the recommendations. For each field in the structure being transformed, it keeps track of the new type that the field will be a part of. If the field is to stay alone, then IPA maps this field to its base type.

Field table update: The WHIRL symbol tables include a field table that contains all fields of all structure types in the program. For a structure type, the TY structure contains a field which is an index into the field table. This index gives the fields belonging to that type. In some cases, the transformation may need to add new fields to existing structure type definitions. One such scenario is when the compiler is peeling type *T* and structure type *S* contains a field that is a *pointer-to-T* (Figure 6). As explained earlier, the analysis phase divides *T*'s fields into two affinity groups AG_1 and AG_2. We peel such a structure type *T* into two sub-structures T_1 and T_2. This part of the transformation updates the fields for *S* by modifying the type of field *p* and then adding a new field pointing to structure T_2.

Updating WHIRL IR: This stage forms the major part of the transformation process. Once the new types have been created and all required type

```
struct S              struct T
{                     {
  // N fields           // AG1 fields
  struct T * p;        // AG2 fields
 // M fields           };
};
```

(a) Before peeling

```
struct S              struct T1             struct T2
{                     {                     {
  // N fields           // AG1 fields         // AG2 fields
  struct T1 * p1;      };                    };
  struct T2 * p2;
  // M fields
};
```

(b) After peeling

Figure 6: Structure peeling of T when S contains a field of type pointer-to-T

definitions have been updated, IPA needs to traverse all PUs and update the WHIRL IR. Let us consider *T* to be the type IPA is transforming. The following are the main updates that need to take place:

- *Statements containing field accesses:* IPA traverses the PUs and updates all statements that access a field in the peeled or split structure. We call the field number in a WHIRL statement a field_id. In statements with field accesses, IPA updates the field_id as well as the field offset. The offset is obtained by traversing the new structure TY to find the field with the proper field_id. The field data structure contains its offset. IPA may also need to update the structure TY being accessed. When structure type *S* contains a field which is a pointer to peeled type *T* (Figure 6), accesses to fields in *S* may also need to change. The type definition of *S* contains fields before and after *p* in Figure 6. For such a type *S*, accesses to all the M fields after *p* need to be updated with their new field_id and offset.
- *New symbols for program variables:* IPA creates new local and global variables to update statements where variables of the transformed type occur. It maintains a map from old ST_IDX (symbol table index) to the new ST_IDX of the corresponding variables. According to this map, it updates statements as it processes the different PUs.
- *Assignment statements:* Statements that assign objects of type *pointer-to-T* need to be replicated to ensure assignment for all the newly created sub-structures. These assignment statements need to be updated with proper type information and new variable names. Assignment statements that need to be handled include both direct assignments (involving STID) and indirect assignments through pointer indirection (involving ISTORE).
- *Function calls to memory management routines:* IPA needs to update calls to memory management routines that allocate or free up memory holding objects of type *T*. For a memory allocation function like *malloc*, the call statement does not tell IPA what type it is allocating. The statement that follows this function call is a store (STID in WHIRL) of a special symbol called the *return value pseudo register* (also called *Return_Val_Preg*). This special symbol holds the value returned by the previous function call. The type of the load of the *Return_Val_Preg* tells IPA the data type being allocated. When this data type is the transformed type *T*, IPA updates the allocation statement as well as the store of the function return value. This update includes replicating the memory allocation statement as well as the STID statement for each sub-structure type created by the transformation. IPA also needs to update the total size of memory being allocated. Typically, allocation function calls use the sizeof operator to tell the function how much memory to allocate. An example function call to allocate memory for holding *N* objects of type *T* and assigning the memory pointer to *p* is:

```
p = (T *) malloc (N * sizeof (T));
```

WHIRL does not have a *sizeof* operator. Hence, the IR will have the *sizeof* operator replaced by the original size of type *T*. IPA updates this with the size of the new structure type being allocated. On the other hand, if *N* is a compile-time constant, then the entire expression will be replaced by the result of the multiplication. In this case, IPA divides the total size (which is the result of the expression *N * sizeof (T)*) by the original size of *T* to determine the number of elements *N* being allocated. IPA then multiplies *N* by the size of the new type to compute the memory that needs to be allocated.

Very often, a memory allocation statement as above is followed by a check to ensure that the function call actually returned usable memory. A typical example is:

```
if (p == NULL)
  exit (1);
```

Once IPA replicates the allocation function call, it also needs to update this conditional check to use all the new variables holding the return values from the memory allocation function calls. Pointer comparisons in other contexts can be adjusted by only comparing pointers to one of the peeled sub-structs. For structure splitting, when the hot root structure is allocated, the compiler also needs to generate code to allocate the cold structure.

5. Performance Results

In this section, we present performance measurements on the CPU2000 and the CPU2006 benchmarks from SPEC [4]. We provide results for the SPEC benchmarks using 64-bit ABI and *-Ofast* option without the use of profile feedback data. The results are from the following system configurations:

- 2.8 GHz AMD Opteron™, with 4 GB of memory and 1 MB of L2 cache
- 3.0 GHz Intel® Xeon® with Core™ microarchitecture, with 4 GB of memory and 4 MB of L2 cache
- 2.0 GHz AMD Barcelona with 8 GB of memory and 512 KB of L2 cache
- 3.4 GHz Intel® Xeon® with EM64T, with 4 GB of memory and 1 MB of L2 cache
- 500 MHz MIPS®-based SiCortex processor with 4 GB of memory and 256 KB of L2 cache

Out of all the CPU2000 and CPU2006 benchmarks, we observe benefits in three benchmarks. They are the CPU2000 FP benchmark 179.art, INT benchmark 181.mcf, and the CPU2006 INT benchmark 462.libquantum. We have not observed any noticeable impact on other benchmarks. The compile-time overhead for the optimizations is negligible. The performance speedups for the three benchmarks on the different sytems are presented in `Table 3`. If a benchmark takes time *T1* to run without the structure layout optimization, and time *T2* to run with the transformation, we calculate the speedup as

(1/T2 – 1/T1) / (1/T1)

We note that out of the three improvements, the AMD Opteron™ system gets the highest speedup on two of them, while the AMD Barcelona system gets the highest speedup on 181.mcf. 181.mcf shows the smallest improvement among the three benchmarks for all five systems. From our experience, we have observed that the structure layout transformations achieve relatively less improvement while compiling for the 32-bit ABI. This is expected, since the smaller pointer size in the 32-bit ABI enables structures containing pointers to be smaller resulting in more cache-friendly behavior.

179.art has a hot structure that is pointed to by a global variable. The benchmark uses a dynamically allocated array of this structure type. The compiler peels the structure type into individual types for the fields of the structure. 181.mcf has a hot structure that is pointed to by several fields in other structures. In addition, the hot structure is a recursive data structure. IPA splits this structure into a hot root structure and a cold child structure that is pointed to by a field in the root. The benchmark 462.libquantum has a hot structure, say *T*, with two fields. The hottest loops of the benchmark access only one of the fields of *T* making it beneficial to peel the hot field out of the structure. This structure *T* is pointed to by a field in another structure, say, *S*. Type *S* uses a dynamically allocated array of objects of type *T*. As a result, the compiler also needs to update the type definition of *S* after peeling type *T*.

The AMD Barcelona system we mentioned above has two quad-core Barcelona chips. Each Barcelona chip has a 2 MB L3 cache shared among the four cores. We ran multiple copies of 462.libquantum simultaneously and measured the performance speedup. The increase in the number of simultaneous runs increases the pressure on the memory system. In addition, the shared L3 cache results in smaller effective cache size for each copy of the run. Hence, as we run more simultaneous copies of the benchmark, we notice increased speedup from the structure peeling transformation. We experimented with 1, 2, and 4 copies of the benchmark, and observed +51% (from Table 3), +69%, and +123% speedup respectively.

6. Related Work

There have been a variety of approaches taken to reduce cache misses by improving data locality. One approach uses loop nest optimizations to modify the access patterns of data arrays [5] and [6]. The work in [7] changes the layout of stack variables, global variables and heap objects to reduce data cache misses. The approach in [8] uses generational garbage collection to ensure that objects affine to each other are placed close to each other. Chilimbi et al [9] describe techniques to improve the locality of dynamically allocated objects. They present two semi-automatic tools called ccmorph and ccmalloc that aid in a cache-conscious allocation. Lattner et al describe automatic pool allocation [10] that uses a context-sensitive pointer analysis to partition heap objects and improve cache performance. Related work on memory allocation and garbage collection to improve memory hierarchy performance has been reported in [11] and [12]. The approach in [13] improves cache performance of dynamically allocated data structures in type-safe programming languages by reordering its data members based on profile feedback. Rubin et al present a profile-based technique that searches the space of possible data layouts [14]. The search proceeds by prototyping candidate data layouts and simulating its performance on a trace of memory accesses.

Benchmarks	AMD Opteron™	AMD Barcelona	Intel® EM64T	Intel® Core™	SiCortex MIPS®
179.art	+169%	+66%	+53%	+60%	+45%
181.mcf	+25%	+35%	+12%	+30%	+7%
462.libquantum	+82%	+51%	+75%	+70%	+69%

Table 3: Performance impact on SPEC benchmarks with structure splitting and structure peeling

Chilimbi et al describe the application of structure splitting and field reordering for Java programs [15]. Truong et al present a field reordering technique called instance interleaving [16]. Zhong et al present reference affinity [17] to measure the "closeness" of data in a reference program trace. Then they use this affinity parameter for array transformation and structure splitting.

Hagog et al in [1] present an implementation of the structure splitting transformation in GCC [18]. Hundt et al [2] describe a framework for structure layout transformations in the HP-UX compiler for Intel® Itanium®. Both these approaches are similar to our framework for the Open64 compiler. While many of the earlier techniques apply only to type-safe languages, these approaches can apply the structure layout transformations even on non-type-safe languages like C and C++. The GCC implementation depends on profile feedback, while our framework works with either statically estimated profiles or dynamic runtime-generated profiles. Hagog et al use a *close proximity graph* and a *field reference graph* to analyze the "closeness" between fields. *Affinity graph*s are used in [2]. While they perform much of the legality and performance analysis in the front-end (FE), we do these tasks in phase IPL which is invoked after the Open64 front-end, and before IPA. By not performing these analyses in the front-end, we only need to implement the analysis once and are able to apply it to all supported languages. Our analysis in IPL can also utilize results of other optimizations that have been done by the Open64 pre-optimizer. We also differ in how the information is analyzed by IPA in assigning "closeness" relationships to structure members.

In addition to a framework for structure layout optimization, Hundt et al present an advisory tool that can reuse compiler analysis data and performance data from runtime to provide advice on improving structure layout. For multi-threaded applications, Raman et al [19] propose a technique for structure layout optimizations for multithreaded applications that reduces false sharing as well as improves spatial locality.

We believe that to maximize the effectiveness of cache locality optimizations, the approaches of loop nest transformations, re-layout of statically and dynamically allocated objects, the structure layout optimizations, as well as techniques to reduce false sharing in multithreaded programs, can be combined without affecting the effectiveness of each other, because they address non-overlapping aspects of this larger problem.

7. Conclusions and Future Work

We have implemented structure layout optimization in the Open64 compiler. The substantial performance improvements that we observe on three benchmarks out of CPU2000 and CPU2006 show that our implementation is highly effective in exploiting the opportunities for structure optimization whenever they arise. Our results also show that the opportunities for structure optimization do not exist in most programs, but when they do, the performance improvement can be quite substantial. Thus, this is an indispensible optimization in modern compilers.

In the process of implementing structure layout optimization in the Open64 compiler, we found that the superior infrastructure of this compiler has allowed our extensions to its data structures to be implemented cleanly and with less effort than we have expected. The entire project took only two man-months. As the compiler is open-sourced, our efforts will benefit the Open64 community as a whole. This work also allows other Open64 developers to build on top of this work and to further improve this optimization.

Several areas hold potential for more interesting work. Our current implementation of static profile estimates can be tuned further. We can relax some restrictions and make the analysis and transformation more general, so that more types can be transformed. This will allow us to apply the transformation to more applications, and analyze the

effects. We can also make the existing field-reordering framework and our new structure layout optimization framework collaborate with each other so as to maximize their combined effects.

8. Acknowledgements

We would like to thank the members of the PathScale compiler group for all their help and encouragement for this project. We would also like to thank the anonymous reviewers for their valuable comments.

References

1. *Cache aware data layout reorganization optimization in gcc.* **Hagog, M and Tice, C.** 2005. Proceedings of the 2005 GCC Developers Summit. pp. 69-92.

2. *Practical structure layout optimization and advice.* **Hundt, Robert, Mannarswamy, Sandya and Chakrabarti, Dhruva.** s.l. : IEEE Computer Society, 2006. Proceedings of the International Symposium on Code Generation and Optimization. pp. 233-244.

3. *Static branch frequency and program profile analysis.* **Wu, Youfeng and Larus, James.** San Jose : ACM, 1994. Proceedings of the 27th annual international symposium on Microarchitecture. pp. 1-11.

4. **SPEC.** *Standard performance evaluation corporation.* [Online] http://www.spec.org.

5. *Improving effective bandwidth through compiler enhancement of global cache.* **Ding, Chen and Kennedy, Ken.** 1, s.l. : Academic Press, Inc., 2004, Vol. 64. ISSN:0743-7315.

6. *A data locality optimizing algorithm.* **Wolf, M and Lam, M.** Toronto : ACM, 1991. Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. pp. 30-44.

7. *Cache-conscious data placement.* **Calder, B, et al.** San Jose : s.n., 1998. Proceedings of the Eightth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII).

8. *Using generational garbage collection to implement cache-conscious data placement.* **Chilimbi, Trishul and Larus, James.** 1998. Proceedings of the 1st International Symposium on Memory Management. pp. 37-48.

9. *Cache-conscious structure layout.* **Chilimbi, Trishul, Hill, Mark and Larus, James.** s.l. : ACM Press, 1999. Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation. pp. 1-12.

10. *Automatic pool allocation: improving performance by controlling data structure layout in the heap.* **Lattner, C and Adve, V.** s.l. : ACM, 2005. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 129-142.

11. *Improving the cache locality of memory allocation.* **Grunwald, D, Zorn, B and Henderson, R.** s.l. : ACM Press, 1993. Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation. pp. 177-186.

12. *Cache performance of garbage-collected programs.* **Reinhold, M.** s.l. : ACM Press, 1994. Proceedings of the ACM SIGPLAN conference on Programming language design and implementation. pp. 206-217.

13. *Automated data-member layout of heap objects to improve memory-hierarchy performance.* **Kistler, T and Franz, M.** 2000, ACM Transactions on Programming Languages and Systems, pp. 490-505.

14. *An efficient profile-analysis framework for data-layout optimizations.* **Rubin, S, Bodik, R and Chilimbi, T.** s.l. : ACM Press, 2002. Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 140-153.

15. *Cache-conscious structure definition.* **Chilimbi, Trishul, Davidson, Bob and Larus, James.** 1999. SIGPLAN Conference on Programming Language Design and Implementation. pp. 13-24.

16. *Improving cache behavior of dynamically allocated data structures.* **Truong, D, N, Bodin, F and Seznec, A.** Washington, DC. : s.n., 1998. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques.

17. *Array regrouping and structure splitting using whole-program reference affinity.* **Zhong, Y, et al.** 2004. Proceedings of the ACM SIGPLAN conference on Programming language design and implementation.

18. **GNU.** *GNU compiler collection.* [Online] http://gcc.gnu.org.

19. *Structure layout optimization for multithreaded programs.* **Raman, E, Hundt, R and Mannarswamy, S.** s.l. : IEEE Computer Society, 2007. Proceedings of the International Symposium on Code Generation and Optimization. pp. 271-282.