TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 10/10 pp231-240
Volume 21, Number 2, April 2016

STrans: A Comprehensive Framework for Structure Transformation

Jiangzhou He*, Wenguang Chen, and Zhizhong Tang

Abstract: Structure Data Layout Optimization (SDLO) is a prevailing compiler optimization technique to improve cache efficiency. Structure transformation is a critical step for SDLO. Diversity of transformation methods and existence of complex data types are major challenges for structure transformation. We have designed and implemented STrans, a well-defined system which provides controllable and comprehensive functionality on structure transformation. Compared to known systems, it has less limitation on data types for transformation. In this paper we give formal definition of the approach STrans transforms data types. We have also designed Transformation Specification Language, a mini language to configure how to transform structures, which can be either manually tuned or generated by compiler. STrans supports three kinds of transformation methods, i.e., splitting, peeling, and pool-splitting, and works well on different combinations of compound data types. STrans is the transformation system used in ASLOP and is well tested for all benchmarks for ASLOP.

Key words: Structure Data Layout Optimization (SDLO); STrans; ASLOP; structure transformation

1 Introduction

As the existence of memory wall, cache utility is very important for performance of applications. Structure in C/C++ is necessary to group related data and improve program readability, but many access patterns to structure fields can cause very poor cache utility. Structure Data Layout Optimization (SDLO) is a prevailing technique to improve cache utility for structure fields access by compiler optimization. SDLO has been proved to be very effective for many applications, and we can usually get 20% to over 100% performance improvement for those applications[1–3].

Splitting, peeling, and pool-splitting are basic structure transformation methods.

- To split a structure, we divide it into two parts, and each of them is a new structure. The first part typically contains all hot fields, and the second one contains cold fields. Besides, we insert a pointer in the first part, which points to the second part.
- To peel a structure, we simply divide it into multiple parts, and each of them is a new structure.
- To pool-split a structure, we divide it into several parts, and each of them is a new structure. We organize them in different memory pools, one for each structure type. We access structure objects via their indices in memory pool.

The three transformation methods are illustrated in Fig. 1.

A structure data layout optimizer typically consists of two main components, a decision maker which figures out the strategy to transform it, and a transformer which transforms the code. There is a broader space

• Jiangzhou He was previously a PhD candidate in Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China, and his contribution to this paper was accomplished before he left Tsinghua University. He is now with Google Inc, 1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA. E-mail: hejiangzhou@gmail.com.
• Wenguang Chen is with Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China, and Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China. E-mail: cwg@tsinghua.edu.cn.
• Zhizhong Tang is with Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: tzz-dcs@tsinghua.edu.cn.
* To whom correspondence should be addressed.
Manuscript received: 2015-05-14; accepted: 2015-06-15

```
{struct foo_t {
    int a, b, c;
};
struct foo_t s[N];
```

(a) Original type definition

```
struct foo_t__cold {
  int b, c;
};
struct foo_t__hot {
  int a;
  struct foo_t__cold *cold_ptr;
};
struct foo_t__hot s__hot[N];
struct foo_t__hot s__cold[N];
```

(b) Split

```
struct foo_t__part1 {
  int a;
};
struct foo_t__part2 {
  int b, c;
};
struct foo_t__part1 s__hot[N];
struct foo_t__part2 s__cold[N];
```

(c) Peel

```
struct foo_t__part1 {
  int a;
};
struct foo_t__part2 {
  int b, c;
};
struct foo_t__part1 *foo_t__part1_pool;
struct foo_t__part2 *foo_t__part2_pool;
size_t s__base_index;
```

(d) Pool-split

Fig. 1 Three basic structure transformation methods.

to explore for the decision maker, so most SDLO related researches focus on it. However, how to build a comprehensive transformer is not trivial. Although we can gain most benefits from transforming arrays of structures, when arrays of one particular structure are transformed, we should also transform other occurrences of such structure, e.g., fields in other transformed or untransformed structures, pointers to them, arrays of pointers, pointers to their arrays, and such compound data types can also appear in other structures, etc. It is not trivial to handle these derived types in a consistent way.

Some existing researchers[1, 4] have already discussed the approach for transformation, but they all have some sorts of limitations, such as failing to support a transformed structure nested in another structure.

In this paper, we present STrans, the transformer in ASLOP[3]. We have made two major contributions:

- We have designed a comprehensive and consistent mechanism to transform a wide range of compound data types.
- We have defined a mini language to specify transformation specification, which can be either generated by SDLO decision maker, or manually tuned. It is helpful for investigating a better transformation scheme for given program.

The rest of the paper is organized as follows. Section 2 presents Transformation Specification Language. Section 3 discusses the transformation mechanism in detail. Section 4 introduces implementation and evaluation briefly. Section 5 summarizes the related works and Section 6 concludes our work.

2 Transformation Specification

STrans does structure data layout transformation according to transformation specification. Transformation specification is described by Transformation Specification Language, a mini domain-specific language designed for STrans.

Figure 2 shows the syntax of Transformation Specification Language. A transformation specification *transform-spec* is made of several transformation directives a.k.a. *transform-directive*, and each of them defines the way to transform one structure. The structure being transformed must not have bit fields. A specific transformation method, i.e., *split*, *peel* or *pool-split*, should be specified for one transformed structure. All transformation methods divide the structure into two or more parts, each of which should be defined by a *transform-part-spec*. A transformation part is simply a list of fields, and each field is either a field in original structure, or a part of a field, if the field has a transformed type.

We have a few restrictions:

- When transformation method is *split*, number of transformed parts must be 2.
- For a field of a structure, if and only if its type's element type is a split or peeled structure, or its type's final target type is a peeled structure, this field can be referenced with a part name. In other words, when transforming a structure, only a field which type meets such requirement can be divided into multiple parts.

Figure 3 shows an example, which transforms two structures, and one of them is used as type of the other's fields.

```
transform-spec ::= transform-directive*
transform-directive ::= 'transform' type-name ':' transform-method
transform-method ::= 'transform-method-name' '{' transform-part-spec* '}'
transform-method-name ::= 'split' | 'peel' | 'pool-split'
transform-part-spec ::= field-list [ ':' part-name ]
field-list ::= field | ( field ',' field-list )
field ::= field-name [ '[' part-name ']' ]
type-name, field-name, part-name ::= identifier
```

Fig. 2 STrans syntax extensions.

```
struct foo_t {
  int a, b, c;
};
struct bar_t {
  int a, b;
  struct foo_t c, d[16];
};
```

(a) Type definition

```
transform foo_t : peel {
  a : hot;
  b, c : cold;
}
transform bar_t : pool-split {
  a, b;
  c, d[hot];
  d[cold];
}
```

(b) Transformation specification

Fig. 3 Sample transformation specification and corresponding type definition.

3 Transformation Mechanism

Transformation Specification Language provides a way to control how to transform specific structures for a program, but it only specifies how to transform each individual types. Transforming the whole program is still not trivial as

- A transformed structure can appear in many different places, including global or local variables declaration, fields of other transformed or untransformed structures and function parameters and returning types.
- A transformed structure type has derived array and pointer types, which can also appear in many different places.
- Different transformation methods can be mixed for one single program.
- Some transformation method needs setup code such as setting the pointer when splitting and calling pool allocation function when pool-splitting.

This section presents the way to transform a whole program with given transformation specification, which is well-defined and consistent.

3.1 Transformation of data types

First, we define a few very fundamental concepts for ease of description in the rest part of the paper.

Definition 1 (Set of data types). $\mathbb{T}$ is the set of all data types. $\mathbb{T}_0 \subset \mathbb{T}$ is the set of all data types that appears in original program.

Definition 2 (Set of transformation specifications). $\mathbb{S}$ is the set of all legal transformation specification for original program.

Definition 3 (Direct transformed structures). $S \in \mathbb{S}$ is a transformation specification. $\mathbb{T}_S^{\mathrm{S}}$, $\mathbb{T}_S^{\mathrm{E}}$, and $\mathbb{T}_S^{\mathrm{P}}$ are the sets of all split, peeled, and pool-split structures, respectively. $\mathbb{T}_S = \mathbb{T}_S^{\mathrm{S}} \cup \mathbb{T}_S^{\mathrm{E}} \cup \mathbb{T}_S^{\mathrm{P}}$ is the set of *direct transformed structures*.

Then we define element type, final target type, and population, three auxiliary functions for the formal definition of data type transformation.

Definition 4 (Element type). Assume $t \in \mathbb{T}$. $E(t) \in \mathbb{T}$ is *element type* of t, which is defined as

$$E(t) = \begin{cases} E(p), & \text{if } t \text{ is array type } p[n]\ (p \in \mathbb{T}_0, n \in \mathbb{N}); \\ t, & \text{otherwise.} \end{cases}$$

For example, $E(\texttt{int}) = \texttt{int}$, and $E(\texttt{struct foo\_t [10][20]}) = \texttt{struct foo\_t}$.

Definition 5 (Final target type). Assume $t \in \mathbb{T}$. $R(t) \in \mathbb{T}$ is *final target type* of t, which is defined as

$$R(t) = \begin{cases} R(p), & \text{if } t \text{ is pointer type } p*\ (p \in \mathbb{T}_0); \\ t, & \text{otherwise.} \end{cases}$$

For example, $R(\texttt{int}) = \texttt{int}$, and $R(\texttt{struct foo\_t **}) = \texttt{struct foo\_t}$.

Definition 6 (Population). Assume $t, p \in \mathbb{T}$. $E(p,t) \in \mathbb{N}$ is the *population* of t in data type p, which is defined as

$$E(p,t) = \begin{cases} 1, & p = t; \\ nE(p,t), & p \text{ is an array } q[n]; \\ \sum_{i=1}^{n} E(q_i,t), & p \text{ is an structure which fields types are } (q_1, q_2, \ldots, q_n); \\ 0, & \text{otherwise.} \end{cases}$$

We say p *contains* t if and only if $E(p,t) > 0$.

Intuitively, population of data type t in another data type p is the overall number of instances of t in an instance of p. It counts instances as array members and

structure fields of p, direct or nested, but not pointer targets. For example, for the types defined in Fig. 3, E (`struct foo_t`, `struct foo_t`) = 1, E(`struct foo_t`, `int`) = 3, E(`struct bar_t`, `struct foo_t`) = 17, E(`struct bar_t`, `int`) = 53. If we add a new field with type `struct foo_t*` to `struct bar_t`, values of E(`struct bar_t`, `struct foo_t`) and E(`struct bar_t`, `int`) will not change.

As definitions of data types are recursive, it is intuitive to define the way to transform data types recursively. However, to do this, we need to resolve the challenge that the dependencies between data types may be circular. For example, in a typical implementation of linked list, the structure of a node, which type is `struct node_t`, has a field which is a pointer to the next node, which type is `struct node_t*`. So definition of `struct node_t` has dependency on the type `struct node_t*`. On the other hand, obviously definition of `struct node_t*` has dependency on type `struct node_t`. These two dependencies are circular. To resolve such circular dependencies, we leverage the fact that C allows target type of a pointer type to be incomplete type[5]. Actually in C/C++ language, the target type of a pointer only needs to be complete when such a pointer is dereferenced. In any circular dependency chains in type definition, there must exist one pointer type which depends on another data type in that chain. To define the way of transforming data types, the definition for pointer types do not need to have dependency on the definition of its target type, which breaks the circular dependency. It works because pointer types are essentially the same thing and only differ in dereference and arithmetic operations, and at the point of these operations the target type should be complete, which is also enforced by the C language specification. Although we do not need the definition of the target type to transform a pointer type, we need to know how parts the target type should be transformed to.

Definition 7 (Number of parts in transformed data type). Assume $t \in \mathbb{T}_0$, $S \in \mathbb{S}$, let $Q_S(t) \in \mathbb{N}$ be number of parts in t's transformed data type, which is defined as

$$Q_S(t) = \begin{cases} Q_S(E(t)), & t \neq E(t); \\ Q_S(R(t)), & t \neq R(t); \\ \text{number of peeled parts of } t \text{ in } S, & t \in \mathbb{T}_S^{\mathrm{E}}; \\ 1, & \text{otherwise.} \end{cases}$$

$Q_S(t)$ tells how many parts the transformed data type of t will have. The definition of $Q_S(t)$ breaks circular dependency as its definition for a structure has no dependency on its definition on its field types.

Now we can formally define how we transform data types.

Definition 8 (Transformed data types). Assume $t \in \mathbb{T}_0$, $S \in \mathbb{S}$, let $\mathcal{F}_S(t) \in \bigcup_{i=0}^{\infty} \mathbb{T}^i$ be transformed types of t under specification S. Note that $\mathbb{T}^0 = \{\epsilon\}$. If $\mathcal{F}_S(t) = \epsilon$, it means the t cannot be transformed under S.

Assume $\boldsymbol{v} \in \bigcup_{i=1}^{\infty} \mathbb{T}_0$, let $\mathcal{L}_S(\boldsymbol{v}) \in \bigcup_{i=0}^{\infty} \mathbb{T}^i$ be transformed types list of $\boldsymbol{v}$ under specification S.

Then we define the value of $\mathcal{F}_S(t)$ for different data types respectively:

- If $\exists p \in \mathbb{T}_0$ s.t., $\mathcal{F}_S(p) = \epsilon$ and t contains p (i.e., $E(p,t) > 0$), $\mathcal{F}_S(t) = \epsilon$.
- If t is a structure with bit field or a union type, and there exists any type p in t's fields that $p \neq \mathcal{F}_S(p)$, $\mathcal{F}_S(t) = \epsilon$.
- If t is a structure type without bit fields and $t \notin \mathbb{T}_S$, assume its fields have types $\boldsymbol{v}$, $\mathcal{F}_S(t)$ is a structure type which fields have types $\mathcal{L}_S(\boldsymbol{v})$.
- If $t \in \mathbb{T}_S$, assume its fields have types $\boldsymbol{v}$, and $\mathcal{L}_S(\boldsymbol{v}) = (t_1, t_2, \ldots, t_n)$. Assume in specification S, t is divided into n parts. We divide t_1, t_2, ..., t_n into n parts according to t's definition in S, and each of them has types $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$.
  - If $t \in \mathbb{T}_S^{\mathrm{E}}$, t_i is a structure which fields have types $\boldsymbol{v}_i$.
  - If $t \in \mathbb{T}_S^{\mathrm{S}}$, by restriction of *split* method, $n = 2$. $\mathcal{F}_S(t) = (t_1, t_2)$, t_1 and t_2 are structures which fields have types $(\boldsymbol{v}_1, t_2{*})$ and $\boldsymbol{v}_2$.
  - Otherwise, $t \in \mathbb{T}_S^{\mathrm{P}}$, $\mathcal{F}_S(t) =$ `size_t`.
- If t is an array type $p[n]$ ($p \in \mathbb{T}_0, n \in \mathbb{N}$). Assume $\mathcal{F}_S(p) = (p_1, p_2, \ldots, p_m)$.
  - If $E(p) \in \mathbb{T}_S^{\mathrm{P}}$, $\mathcal{F}_S(t) =$ `size_t`.
  - Otherwise, $\mathcal{F}_S(t) = (p_1[n], p_2[n], \ldots, p_m[n])$.
- If t is a pointer type p^* ($p \in \mathbb{T}_0$).
  - If $E(p) \in \mathbb{T}_S^{\mathrm{S}}$, $\mathcal{F}_S(t)$ is a pointer type to the first type of $\mathcal{F}_S(p)$.
  - If $E(p) \in \mathbb{T}_S^{\mathrm{P}}$, $\mathcal{F}_S(t) =$ `size_t`.
  - Otherwise, $\mathcal{F}_S(t)$ is $Q_S(t)$ pointer types, each of which points to one part of $\mathcal{F}_S(p)$.
- If t is a function type which parameter

types are $q_1, q_2, \ldots, q_n$ and returning type is p. Assume $\mathcal{F}_S(p) = (p_1, p_2, \ldots, p_m)$, $\mathcal{L}_S((q_1, q_2, \ldots, q_n)) = (r_1, r_2, \ldots, r_l)$. Then $\mathcal{F}_S(t)$ is a function type which parameter types are $(r_1, r_2, \ldots, r_l, p_2*, \ldots, p_m*)$ and returning type is p_1.

- Otherwise, $\mathcal{F}_S(t) = t$.

The value of $\mathcal{L}_S(\boldsymbol{v})$ is determined by trasnformed types of each element of $\boldsymbol{v}$ with all redundant pool index eliminated. To be precise, assume $\boldsymbol{v} = (p_1, p_2, \ldots, p_n)$,

$$\mathcal{L}_S(\boldsymbol{v}) = \begin{cases} \mathcal{F}_S(p_1), & n = 1; \\ \mathcal{L}_S((p_1, p_2, \ldots, p_{n-1})), & \\ \quad E(p_n) \in \mathbb{T}_S^{\mathrm{P}} \text{ and } E(p_n) \in \bigcup_{i=1}^{n-1} \{E(p_i)\}; & \\ (\mathcal{L}_S((p_1, p_2, \ldots, p_{n-1})), \mathcal{F}_S(p_n)), & \\ \quad \text{otherwise}; & \end{cases}$$

The most basic property of $\mathcal{F}_S$ is to guarantee each piece of data which is accessible from $t \in \mathbb{T}_0$ also accessible from $\mathcal{F}_S(t)$, either directly or through memory pool.

When transforming pointers to split structures, we only need to keep pointer to its hot part, as we can get the pointer to its second part from the hot part.

When a pool-split structure becomes array element or multiple fields in other structures, we only need to create one index with type `size_t`, as these elements will be allocated in the memory pool by single allocation call, so their indices are continuous and each element has a fixed offset to the base index.

$\mathcal{L}_S$ is a function to describe how to transform a list of objects. In our view, variables in a global or local scope, fields in a strcture or class, and parameters of a function are all lists of objects, and they can be transformed in a unified way, which is defined by $\mathcal{L}_S$.

Figure 4 shows an example of transformed types.

3.2 Restrictions

The type transformation function $\mathcal{F}_S$ is well defined for most possible data types except for structures with bit fields or unions which need to be transformed. For such data types t which cannot be transformed, $\mathcal{F}_S(t) = \epsilon$.

If the program contains any unsafe pointer cast, we cannot transform it. Some standard C functions such as `malloc`, `calloc`, `realloc`, `free`, `memcpy`, `memmove`, and `memset` are exceptions as we handle them specifically.

```
struct far_t {
  int a;
  struct bar_t b, c;
  struct bar_t d[256];
  int e;
};
```

(a) Additional type definition

```
struct foo_t__hot {
  int a;
};
struct foo_t__cold {
  int a, c;
};
struct bar_t__part1 {
  int a, b;
};
struct bar_t__part2 {
  struct foo_t__hot c__hot;
  struct foo_t__cold c__cold;
  struct foo_t__hot d__hot[16];
};
struct bar_t__part3 {
  struct foo_t__hot d__cold[16];
};
struct bar_t__part1 *foo_t__part1_pool;
struct bar_t__part2 *foo_t__part2_pool;
struct bar_t__part3 *foo_t__part3_pool;
struct far_t__new {
  int a;
  size_t bar_t__base_index;
  int e;
};
```

(b) Transformed types

Fig. 4 Sample transformed types. Definition and transformation specification of `bar_t` are in Fig. 3.

Another restriction is that we cannot transform the program if the whole source code is only partly visible and pointer types which needs to be transform appears in the interface between the visible part and the invisible part.

3.3 Transformation of variable list

We need to transform variable lists in each scope. The way to transform the variable list defined in any particular scope is essentially the same as transforming the fields of a untransformed structure. For transformation specification $S \in \mathbb{S}$, for any scope with variables of types $\boldsymbol{v} \in \bigcup_{i=0}^{\infty} \mathbb{T}_S^{+i}$, the types of transformed variables are determined by $\mathcal{L}_S(\boldsymbol{v})$.

3.4 Transformation of code for execution

3.4.1 Access to transformed types

Since Definition 3.1 guarantees each data accessible from any original data type also accessible from its transformed types, it is obviously possible to transform load and store operations for the original data types to transformed ones.

Sometimes we have to introduce new pointer dereference, i.e., when we need to access the cold part of a split structure via its pointer stored in its hot part, and when we need to access a pool-split structure object

via memory pool. However, we still want to avoid new pointer dereference whenever possible. For example, to access fields in the cold part of split structures, we can still access it directly as long as in the original program such structure object is accessed directly without using a pointer. If the original program accesses a structure by pointer and such structure is pool-split, the transformed one accessed it by memory pool base pointer and index, which is not considered as an additional level of deference. However, it is inevitable to introduce new pointer dereference while in the original program a split structure object is referenced by a pointer to it, or an object of a pool-split structure is accessed without a pointer in the original program.

3.4.2 Assignment operations

Assignment operation of a structure object is equivalent to assignment of each fields of such object. So transformation of load and store operations of their fields covers this situation.

Sometimes the original program uses bulk memory operations such as `memcpy`, `memmove`, and `memset`, we need to transform them carefully. If the target type has been transformed into multiple parts, the bulk memory operations should be also transformed into multiple parts. However, if the target type contains object of split structures, i.e., it has a non-zero population of such split structure, we need to make sure the pointer to the cold part which is stored in the hot part not copied, moved or set. To do this, we transform such `memcpy`, `memmove`, and `memset` to loops of single element operations first.

Note that we do not support `memcmp` and `memchr` when it is called for a structure type which contains a transformed structure.

3.4.3 Function definitions and function calls

As already stated in $\mathcal{F}_S$, function types is transformed, so function definitions and function calls will be transformed accordingly.

If the returning type original function is transformed to multiple types, all but the first type is transformed to parameters with their pointer type. We need to transform the returning statement to writing return values by incoming pointers. We also need to transform corresponding function call as well by passing pointers to the transformed objects which takes the return value.

3.4.4 Transformation of pointer operations

Figure 5 shows how we transform pointer operations. For pointers to split and pool-split structures, it is much more straightforward as the pointer is transformed to a single object, either a pointer or an index. For pointers to peeled structures, situations become more complex as there are multiple pointers after transformation. The main principle is that for any expression with a pointer result type, we need to duplicate such expression into multiple ones. For all other expressions, we replace the original pointer by the pointer to the first peeled part. Actually for all possible situations in the latter case, we can use pointer to any part, which will yield the same result.

	split	peel	pool-split
`p=q`	`p1=q1` `p2=q2`	`p1=q1`	`pi=qi`
`p=NULL`	`p1=NULL` `p2=NULL`	`p1=NULL`	`pi=0`
`p+n`	`p1+n` `p2+n`	`p1+n`	`pi+n`
`p-q`	`p1-q1`	`p1-q1`	`pi-qi`
`p=q`	`p1=q1`	`p1=q1`	`pi=qi`
`p<q`	`p1<q1`	`p1<q1`	`pi<qi`
`p==NULL`	`p1==NULL`	`p1==NULL`	`pi==0`

Fig. 5 Transformation pattern of pointer operations. Suppose `p` and `q` are pointers to a transformed structure. `p1`, `p2`, `q1`, and `q2` are transformed pointers to their split or peeled parts. Without loss of generality, we suppose the original structures is peeled into two parts. `pi` and `qi` are their transformed indices. `n` is some integer variable.

3.4.5 Memory allocation and releasing

We also need to transform code of allocating, reallocating, and deallocating memory for objects of transformed types. In C, this indicates that we need to transform call-sites of `malloc()`, `calloc()`, `realloc()`, and `free()`, which parameter type is pointer to transformed types or returning type is stored in pointer to transformed types.

We need to replace these function calls with multiple ones for split and peeled structures. We need to remove these function calls for pool-split structures and then add initialization code for it. Besides, if the target data type contains a split or pool-split structure, we also need to add initialization code, which will be covered in Section 3.4.6 in details.

3.4.6 Initialization and finalization code

For untransformed C data types, a variable is uninitialized after allocated on stack and heap, and a variable with static storage duration is zero-initialized by default. But we need to generate code to initialize pointers to cold part of split structures and memory pool

indices.

For any data type containing split structures, we need to generate code to initialize the cold part pointer stored in the hot part.

For any variable which contains pool-split structures, we need to insert memory pool API invocation to initialize its indices fields after it is allocated and releasing corresponding elements in the memory pool before it is released. For each pool-split structure, we use its population in all objects in current scope (for heap allocation, all objects allocated by one `malloc()` call are considered to be in the same scope) as number of elements to be allocated in memory pool. In other words, we merge all function call to allocate new pool elements for the same scope, and add the offset of each object with the explicit index field which is set by return value of that function call to set indices for these objects.

For variables on stack, the initialization and finalization code is inserted at the entry point and exit of the function. For function parameters, they are also allocated on stack and they should be allocated and released by caller, so the initialization and finalization code is inserted before and after the function call in caller.

For variables allocated on heap, those codes come with `malloc()` and `free()` function calls.

For variables with static storage duration defined in each translation unit, we need to make sure the initialization code is executed once and only once after the program starts and before they are first referenced.

4 Implementation and Evaluation

4.1 Implementation

We have implemented STrans in Open64[6] compiler. We do the transformation in backend, as the decision maker needs to collect information in the inter-procedural analysis phase and STrans needs to run after it. The implementation of STrans is about 6000 lines of C++ code.

Figure 6 shows data flow of the transformer. It consists of the following components.

4.1.1 Transformation specification parser

It parses the transformation specification. The parser is based on flex[7] and bison[8]. It generates Transformation Specification IR, which is essentially a three-tier tree for each transformed structure, i.e., the first tier is the root which has pointer to the original type in the type table and the transformation method, the second tier is for each part and notes their names if specified and the third tier is the pointer to fields of the original type in the type table.

4.1.2 Type transformer

It creates new types according to original types and transformation specification, and maintains a mapping between original types and new types.

It creates new types in two steps. First, it creates entries for new types without defining them. Q_S defined

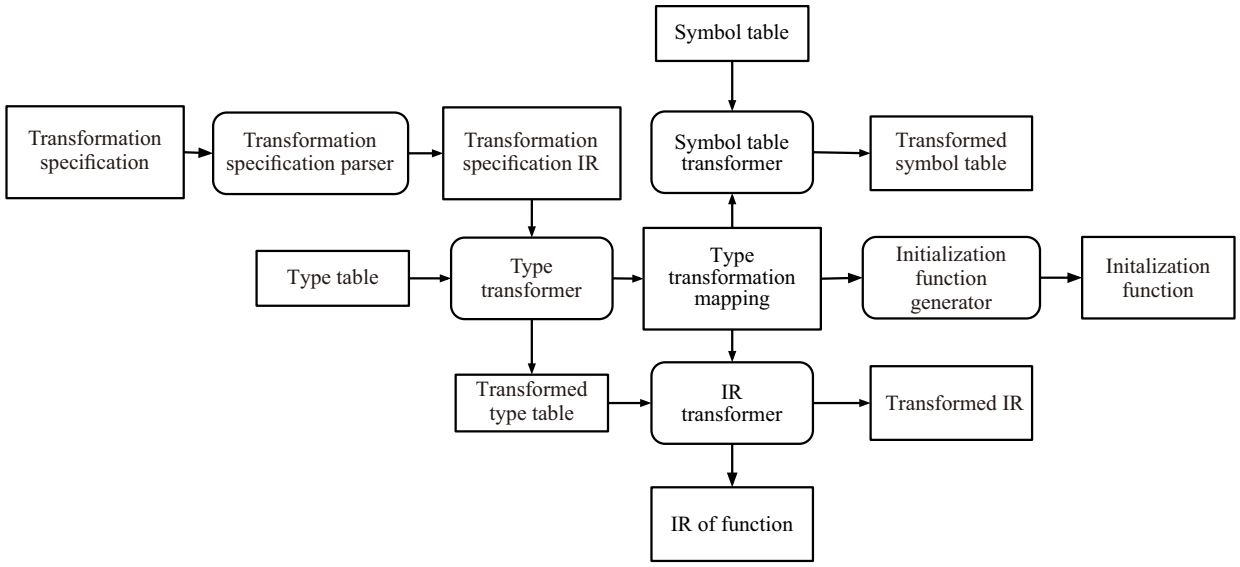


Fig. 6 System data flow. Data are represented by rectangles and components of the transformer are represented by rounded rectangles.

in Definition 3.1 is the most important tool for creating new types. Then it defines the new types according to F_S defined in Definition 3.1.

4.1.3 Symbol table transformer

It transforms global and local symbol tables to create new variables if their original data type is transformed to multiple ones, and change data types for variables with other transformed data types.

4.1.4 IR transformer

It transforms IR of each Program Unit (PU) as what Section 3.4 said. The most tricky part is to understand data access pattern expressed by IR, as in IR access to a field of a structure object is expressed by low level information such as its base address and offset.

Let's use the data types defined in Fig. 3 as an example again. Suppose in the target ABI size and alignment of `int` are 4, so structure `foo_t`'s size is 12 and its alignment is 4, and the offset of field `d` in structure `foo_t` is 20. Suppose `s` is an object of structure type `bar_t` and `i` is an integer variable. In IR, access to `s.d[i].b` is using address of `s` as base and using $20+12i+4$ as offset. All constants in IR are aggregated, so the offset looks like $12i+24$. We need to recover the actual field and array indices from IR.

To demonstrate how we solve the problem, considerring the target type of the base address is $t \in \mathbb{T}$ and the offset expression has the form $c_0 + \sum_{i=1}^{N} c_i V_i$ where $c_0 \in \mathbb{N}$, $c_i\ (0 < i < N) \in \mathbb{Z}$, and $V_i\ (0 < i < N)$ is any variable in the PU. The algorithm to recover the field and array indices is recursive.

- If t is a structure type, among all its fields we find out the one with maximum offset and less than c_0, which is the field to access. Assume its offset is d, $(c_0 - d) + \sum_{i=1}^{N} c_i V_i$ is the remainder offset expression to access that particular field with its data type.
- If t is an array type, assume its size is s, the index to access it is $\left\lfloor \frac{c_0}{s} \right\rfloor + \sum_{i=1}^{N} \left\lfloor \frac{c_i}{s} \right\rfloor V_i$, and the remainder offset expression to access its element by its type is $\left(c_0 - \left\lfloor \frac{c_0}{s} \right\rfloor s\right) + \sum_{i=1}^{N} \left(c_i - \left\lfloor \frac{c_i}{s} \right\rfloor s\right) V_i$.

In this way, we recover the fields to access structures and the indices to access array elements from low level offset expression. We call such high level fields and indices information by accessing path. Based on the mapping between original data types and transformed data types, we can transform these accessing paths for original data types to new accessing paths for transformed data types. Finally, we generate IR to replace the original IR based on the new accessing paths, which is essentially the inverse procedure of the way to recover accessing paths, but much more straightforward.

Beside transforming IR, we also need to insert code to initialize and finalize some transformed local variables as stated in Section 3.4.6. It is a little bit tricky to do this for local static variables. To do this, we create static boolean variables for each function with static variables, and at the beginning of the function we insert code to check it and when the variable has a false value we make it execute initialization code for static variables and set the flag to true.

4.1.5 Initialization function generator

It generates initialization functions for each translation unit. The main purpose of such functions is to initialize transformed variables in global scope. To make sure such functions are executed before `main()` is executed, we add such functions into ".ctor" section, so the linker will put all these function into a list which will be executed when the program starts.

4.2 Evaluation

As STrans is a part of ASLOP, all benchmarks evaluated in ASLOP[3] are well transformed by STrans, including 179.art and 181.mcf from SPEC CPU 2000[9], 462.libquantum, 429.mcf, and 472.moldyn from SPEC CPU 2006[10], em3d, health, treeadd, and tsp from Olden benchmarks[11]. For each benchmark, we have applied each transformation method. All benchmarks work correctly after transformation.

5 Related Work

Ding and Zhong[4] investigated on how to use memory pool to split structures. However, their approach has obvious limitations. For example, to handle the case when a pool-split object is nested into another structure, when the parent structure is pool-split, they convert the child field into a pointer, which makes the storage of objects for the same pool-split different by context. Moreover, it does not support a pool-split object nested

in an unsplit object.

Hundt et al.[1] proposed splitting and peeling structures. But they do not support to split any structure which is nested in another type.

Zhao et al.[2] proposed an address-arithmetic-based transformation method, which computes address of pointer to cold parts by pointer arithmetic. It also has some extra restrictions, for example, it enforces single-instantiation restriction, which refuses to transform those dynamic allocated arrays that may be instantiated at different sites on different branches. And it does not mention how to handle the case that a transformed object is nested in another structure.

There are many papers focusing on making decision on a good SDLO or general data layout strategy[2, 3, 12–16].

STrans is used as the transformer in ASLOP[3], which makes SDLO decision based on both intra-instance affinity and inter-instance affinity.

To our best knowledge, there is no existing work which designed an SDLO transformation mechanism with no more restrictions than STrans.

6 Conclusion

In this paper, we present STrans, a comprehensive framework for SDLO. STrans has a definite and consistent mechanism to transform most possible compound data types by splitting, peeling or pool-splitting. We have also designed Transformation Specification Language, which is a reasonable interface of STrans for both human tuned and machine-generated transformation strategy. STrans already makes ASLOP happen and we believe it will also benefit future SDLO research and system implementation.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61133006) and the National High-Tech Research and Development (863) Program of China (No. 2012AA010901).

References

[1] R. Hundt, S. Mannarswamy, and D. R. Chakrabarti, Practical structure layout optimization and advice, in *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006)*, 2006, pp. 233–244.

[2] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral, Forma: A framework for safe automatic array reshaping, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 1, article no. 2, 2007.

[3] J. Yan, J. He, W. Chen, P. Yew, and W. Zheng, ASLOP: A field access affinity-based structure data layout optimizer, *SCIENCE CHINA Information Sciences*, vol. 54, no. 9, pp. 1769–1783, 2011. doi:10.1007/s11432-011-4265-0.

[4] C. Ding and Y. Zhong, Predicting whole-program locality through reuse distance analysis, in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003, pp. 245–257.

[5] ISO/IEC JTC 1, C99 standard, http://www.openstd.org/JTC1/SC22/WG14/, 2015

[6] Open64 compiler, http://www.open64.net, 2015

[7] J. Levine, *flex & bison: Text Processing Tools*. O'Reilly Media, 2009.

[8] C. Donnelly and R. Stallman, Bison: The Yacc–Compatible parser generator, Free Software Foundation, 1992.

[9] SPEC CPU 2000 Benchmark Suite, http://www.spec.org, 2015

[10] SPEC CPU 2006 Benchmark Suite, http://www.spec.org, 2015

[11] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, Supporting dynamic data structures on distributed-memory machines, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 2, pp. 233–263, 1995.

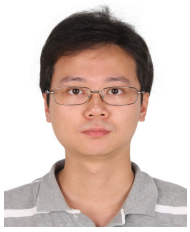
[12] T. Kistler and M. Franz, Automated data-member layout of help objects to improve memory-hierarchy performance, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 3, pp. 490–505, 2000

[13] T. M. Chilimbi, Efficient representations and abstractions for quantifying and exploiting data reference locality, in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2001, pp. 191–202.

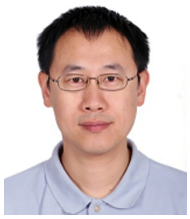
[14] Y. Zhong, M. Orlovich, X. Shen, and C. Ding, Array regrouping and structure splitting using whole-program reference affinity, in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004, pp. 255–266.

[15] E. Petrank and D. Rawitz, The hardness of cache conscious data placement, in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 101–112.

[16] Y. Zhong, X. Shen, and C. Ding, A hierarchical model of reference affinity, in *Languages and Compilers for Parallel Computing, 16th International Workshop, LCPC 2003*, 2003, pp. 48–63.



Jiangzhou He received the PhD degree in computer science from Tsinghua University in 2007. Now he is a software engineer in Google Inc. His research interests include parallel and distributed computing, compiler technology, and programming models. He is a member of CCF.



Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University in 1995 and 2000, respectively. He was the CTO of Opportunity International Inc. from 2000 to 2002. Since January 2003, he joined Tsinghua University. He is now a professor in the Department of Computer Science and Technology at Tsinghua University. His research interests include parallel and distributed computing, programming model, and mobile cloud computing. He is a member of CCF, ACM, and IEEE.

Zhizhong Tang received the BS degree from Tsinghua University in 1970. He is now a professor in the Department of Computer Science and Technology at Tsinghua University. His research interests include compiler technique and CMP cache optimization. He is a member of CCF.