The new intraprocedural Scalar Replacement of Aggregates

Martin Jambor
SUSE Labs
mjambor@suse.cz

Abstract

Scientific literature does not usually describe Scalar Replacement of Aggregates (SRA) in great depth. Indeed, the basic idea is simple but in practice there are surprisingly many issues that are minor on their own but which make the transformation unexpectedly complex on the whole. Some of these difficulties are imposed by the intermediate language, others are direct consequences of the nature of the problem which itself is less straightforward than one might assume. We list the goals SRA is expected to achieve and pitfalls it tries to avoid in order to describe what people expect from SRA. We then outline the implementation structure in general, pay special attention to the less clear parts and show what heuristics are employed to make scalarization decisions. We also discuss some of the more interesting related bugs. Finally, we briefly cover SRA development in GCC 4.6 and potential future improvements.

1 Introduction

Scalar Replacement of Aggregates (SRA) creates new scalar variables for parts of aggregates which cannot be aliased. The chief motivation is to make optimizations which are applicable only to scalar variables also work on values stored in such aggregates and perhaps to eliminate some loads from memory. Scientific literature (e.g. [2, 3]) usually does not deal with this transformation in much more detail than stating the above, perhaps with an example or two. On the other hand, when we started to re-implement this transformation for GCC 4.5, we hoped not only to make it more capable but, perhaps even more importantly, also simpler. We deemed the previous implementation unnecessarily complex, mainly because it was trying to be perhaps too clever. We have successfully achieved the first goal but the pass is certainly more complex than we had originally hoped it would be and the number of reported bugs and other complaints has also been surprisingly high.

```
struct S
{
  int i;
  int j;
};
```

```
int foo1 (bool b)
{
    struct S s;
     s.i = 2;
    if (b)
        s.i = 1;
    else
        s.i--;
    return s.i;
}
```

Figure 1: A simple opportunity to do SRA.

There are two reasons for this. First, *gimple* intermediate language constraints present us with many details in which the proverbial devil can lurk, and second, people have learned to expect SRA to do a fair bit more than the simple replacement.

In this paper we therefore intend to present the problem itself more thoroughly. We will start by enumerating what the current SRA implementation is expected to do, including some of the new features and potential undesirable situations it needs to avoid. The next section then provides an overview of the implementation and looks in more detail at the main data structures and some parts of the code which are necessary to achieve some of the stated goals. Finally, we will discuss what development is new in 4.6 and what the potential improvements in the future might be.

2 Goals

This section lists all the tasks SRA is supposed to perform. Some of the items might be rather obvious, nevertheless we specifically wanted to make the list as complete as possible. When looking at the examples presented here, please keep in mind that the control flow is often more complicated, that SRA operates on single statements and uses no data flow analysis to make decisions. The goals are:

```
struct SA
{
  short a[1];
  int b[4];
};
```

```
int foo2 (int i, int v)
{
   struct SA s;

   s.a[i] = 2;
   s.b[2] = v;
   return s.a[0]+s.b[2];
}
```

Figure 2: Simple SRA of array elements.

i) Create replacements of structure[1] fields

As we have already stated, the basic tasks of SRA is to try to replace uses of scalar parts of local non-aliased aggregates with new variables whenever there is a chance other optimizations might then leverage on them. For example in figure 1, SRA has to replace all occurrences of `s.i` with a new scalar variable which will allow SSA based constant propagation to optimize the whole function to a simple return of constant 1.

ii) Create replacements of array elements (PR 13952)

SRA should be able to replace even parts of arrays. The main difference is that all used indices have to be known so that we can distinguish in between individual elements in all accesses to an array. Perhaps slightly non-intuitively, an index into a one element array is always known, even if it looks variable. Such accesses are special because the reference expression cannot be directly moved around the function or used when generating debug information for the replacement. Both types of such accesses are illustrated in figure 2.

iii) Handle uses of whole aggregates

An obvious correctness requirement is that when an aggregate is used as a whole, values of all replacements must be stored to their original place and when an entire aggregate is assigned a value, the replacements must load their respective values from it. This is also true for aggregates within aggregates (see figure 3 for an example).

iv) Treat aggregate assignments differently

Assignments in between aggregates need to be treated with more care. If there are replacements created for aggregates on both sides of the assignment, we would like to generate assignments in

[1] In this paper, we use the terms *structure* and *record* interchangeably.

```
struct S
{
  int i;
  int j;
};
```

```
struct S2
{
  struct S s;
  double d;
};
```

```
int foo3 ()
{
  struct S s;
  struct S2 r;

  /* computation with s.i and s.j */
  r.s = bah (s);
  /* computation with parts of r */
}

int foo3_sra ()
{
  struct S s;
  struct S2 r;

  /* computation with s$i and s$j */
  s.i = s$i;
  s.j = s$j;
  r.s = bah (s);
  r$s$i = r.s.i;
  r$s$j = r.s.j;
  /* computation with r replacements */
}
```

Figure 3: Aggregate accesses and replacements.

between the corresponding replacements, bypassing the original aggregates (see example 4). Even when that is not the case we should still attempt to load all replacements of the left hand side from the aggregate on the right hand side or vice versa. Moreover, SRA can eliminate the original assignment altogether if:

- replacements on the left had side cover all data transferred in the assignment, or
- parts of the aggregate on the right hand side which are not covered by replacements of the aggregate on the left hand side contain only uninitialized (undefined) data, or
- parts of the aggregate on the left hand side which are not covered by its replacements are never used.

v) Remove loads of uninitialized data

Even when there are no replacements at all created for parts of aggregates that are copied or loaded

```
int foo4 ()
{
  struct S s;
  struct S2 r;

  /* computation with s.i and s.j */
  r.s = s;
  /* computation with parts of r */
}

int foo4_sra ()
{
  struct S s;
  struct S2 r;

  /* computation with s$i and s$j */
  r$s$i = s$i;
  r$s$j = s$j;
  /* computation with r replacements */
}
```

Figure 4: Aggregate assignments and replacements.

in an assignment statement, SRA should remove the load altogether if the right hand side is never written to and therefore contains uninitialized data. Apart from removing unnecessary code, this transformation enables us to issue warnings when the assignment type is scalar (PR 44133) and we must take special care to make the warning intelligible for the user.

vi) Perform a pseudo copy propagation

Upon encountering an assignment between two aggregates which are eligible for SRA, the pass considers creating replacements of the left hand side which would correspond to replacements of the right hand side even if there are no such accesses in the original function. This makes the ordinary scalar copy propagation work on values originally stored within aggregates. Because of the special assignment handling discussed above, the original aggregates might even be removed altogether like in figure 5.

vii) Scalarize simple structures away (PR 42585)

In fact, SRA will actively replace small simple structures with replacements of all of their scalar contents if that makes them disappear – they cannot disappear if they are used in other statements than assignments. A structure is considered simple if it is of `RECORD_TYPE` and its fields are all either non-bit-field scalars or other simple structures. Structures with bit-fields are not considered because currently their forced scalarization might lead to worse final code (PR 45144). This process, called *total scalarization*, can lead to substantial savings of stack space and allow better code generation later in the compilation pipeline. See PR 42586 for an example similar to the one in figure 6.

```
int foo5 (int v)
{
  struct S x,y,z;
  x.i = v;
  y = x;
  z = y;
  return z.i;
}
```

```
int foo5_sra (int v)
{
  int x$i,y$i,z$i;
  x$i = v;
  y$i = x$i;
  z$i = y$i;
  return z$i;
}
```

Figure 5: SRA copy-propagation enablement.

```
void foo6 (struct S *p)
{
  struct S x,y;
  x = *p;
  y = x;
  y.i++;
  *p = y;
}

int foo6_sra (struct S *p)
{
  int x$i, x$j, y$i, y$j;
  x$i = p->i; x$j = p->j;
  y$i = x$i; y$j = x$j;
  y$i++;
  p->i = y$i; p->j = y$j;
}
```

Figure 6: SRA total scalarization.

viii) Optimize unions too (PR 32964)

Unions are aggregates too and even though not all of them can be scalarized, they do also present a number of opportunities for SRA that should be exploited. A union component can preclude scalarization of an aggregate if any two accesses to different union fields *partially overlap* (i.e. each contains some data not included within the other). Moreover, when scalarizing a union, SRA has to decide what the type of each created replacement will be.

Naturally, SRA should be able to analyze and create scalar replacement for aggregates consisting of any combination of structures, arrays and unions with accesses that comply with the condition set in the previous paragraph. We should also note that type casting (VIEW_CONVERT_EXPRs) behave very similarly to unions and pose similar problems.

ix) Handle vector and complex type

Vectors and complex numbers are considered scalar types in gimple but standalone variables of this type may or may not be registers (DECL_GIMPLE_REG_P) depending on whether they are always modified in their entirety or by parts. When SRA creates a complex or vector variable it has to mark it as a register if possible.

x) Be able to analyze and produce MEM_REFs

In order to work well in GCC 4.6, SRA must be able to understand MEM_REFs into local aggregates. These references can also behave like type casts and ignore the type definition of the aggregate. Additionally, SRA should produce MEM_REF rather than other memory references whenever possible.

xi) Avoid creating unnecessary (PR 40744) or harmfully excessive statements (PR 44423)

There are SRA opportunities that are not beneficial and therefore should not be exploited. Often SRA cannot determine this and then optimistically creates replacements but there are many cases when it is evident that any created replacements are pointless and the statements to load and store them will be immediately removed by copy propagation or dead code elimination. For example when there is only one store to a particular piece of an aggregate and then the aggregate is used as a whole (in a way that prevents total scalarization) or when there is only one read to a portion of it like in figure 7, creating replacements will only add extra variables and statements that will be of no benefit as far as final code is concerned, following passes will remove them and SRA should therefore refrain from creating them. Nevertheless, note that this prevents us from issuing some uninitialized variable warnings, like in function foo9 in the example.

In addition to statements that are useless, wrong SRA decisions can also lead to serious code slowdowns. PR 44423 was about a union in between an SSE vector and an array of its elements. The elements were used to initialize the data which were then accessed through the vector type in a hot loop. The elements were instantiated as separate scalar replacements, which were stored into the original union each time the vector was passed to a builtin arithmetic function (to satisfy requirement iii). Therefore a number of slow stores were introduced to each iteration of the loop and since no pass at the moment hoists loop invariant loads or stores, this led to severe slowdowns. In order to avoid this, SRA currently does not create replacements for scalars that are within another scalar in the original aggregate.

```
void foo7 (void)
{
  struct S x;
  x.i = 1;
  x.j = 2;
  bah (x);
}
```

```
int foo8 (struct S x)
{
  return x.i;
}

int foo9 (void)
{
  struct S x;
  return x.i;
}
```

Figure 7: Examples of situations when SRA is not beneficial.

Finally, SRA should produce valid gimple. This is perhaps an obvious requirement for a gimple pass but there is enough code in SRA dealing with various corner cases that it is certainly worth mentioning. The obvious source of potential violations of gimple grammar and type rules are unions and type casts in the original function. SRA deals with these issues usually by generating VIEW_CONVERT_EXPRs or, when that is not possible (e.g. in a call statement) by using the original aggregate to do the type punning. Nevertheless, there were also other reasons why we ran into gimple verification failures in the past. One is that gimple imposes tougher restriction on uses of register types. For example when a complex value is assigned in between two new replacements which are not registers (see goal ix), an intermediate temporary has to be created and the store must be done in two steps through the new temporary in order to satisfy the gimple rule that a register type value must be loaded into a register before it is used. Removing a scalar assignment of dead code (goal v) might lead to an SSA name without a definition, and so on, and so forth.

3 Implementation overview

SRA runs twice when optimizations are enabled, the first time as an early optimization before IPA passes and then during the regular passes after IPA transformations are executed. Both times it runs after forward propagation which removes `TREE_ADDRESSABLE` flag from some of local aggregates. Both runs are nearly identical, consisting of four fairly separate stages:

3.1 Identifying candidates

SRA pass keeps a bitmap of potential candidates `candidate_bitmap` and never creates any replacements for any aggregate which is does not have the corresponding bit in this bitmap set. It starts by setting these bits for all aggregates which satisfy the basic requirements. The entire test can be found in function `find_var_candidates`, the main conditions are that candidates must not live in memory, be volatile or have volatile components.

3.2 Function body scan

If there are any candidates, SRA proceeds to scan all instructions of a function and analyzes them one by one. This part of the pass is shared with IPA-SRA [1] which also checks access through pointer parameters which the intraprocedural SRA ignores. Each instruction is examined for accesses to aggregate candidates. If such an access somehow rules out creating any scalar replacement for a given aggregate, its bit in candidate bitmap is cleared. We say that such a candidate is *disqualified*. A good example of a reason for disqualification is a use of an aggregate in a `GIMPLE_ASM` statement. Otherwise, the pass creates an `access` structure for every load or store, recording the aggregate (called `base`), offset, size,and a number of flags, most prominently whether it represents is a read or write reference, is a part of an assignment, whether the access represents and unscalarizable region (result of variable array accesses) or modifies only a part of a complex number or a vector (to be able to achieve goal ix). The base, offset and size is determined using `get_ref_base_and_extent` functions which allows us to handle unions, typecasts (goal viii) and `MEM_REF`s (goal x). SRA also creates a vector of pointers to access structures for each aggregate eligible for scalarization.

```
union U                 unsigned bar1 (void)
{                       {
  signed i;               struct S s, r;
  unsigned x;
};                        s.u.i = -4;
struct S                  s.p = &g1;
{                         r = s;
  union U u;              s.p = &g2;
  void *p;                baz(r);
};                        return S.u.u;
                        }
```

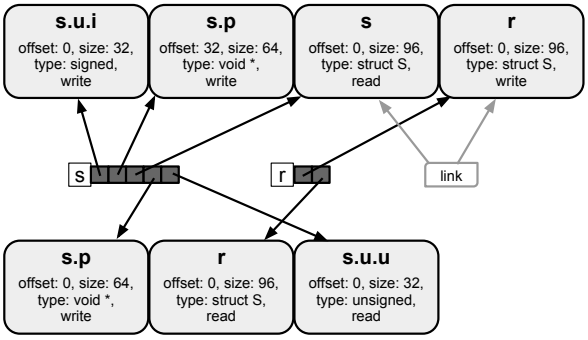

Figure 8: Structures created by SRA after the initial function scan. Additionally, `s` would be marked in `should_scalarize_away_bitmap` and `r` in `cannot_scalarize_away_bitmap`.

SRA takes special care when analyzing aggregate type assignments. It marks each aggregate occuring on the right hand side in `should_scalarize_away_bitmap` because this assignment might be removed if the aggregate is totally scalarized (goal vii). It also creates a small link structure between the accesses representing both sides of the assignment if both sides are candidates for SRA in order to perform the pseudo copy propagation (goal vi). On the other hand, if there is an aggregate access in a statement other than an assignment, SRA marks the aggregate in `cannot_scalarize_away_bitmap` to record that total scalarization of it would be pointless because the variable simply cannot disappear.

3.3 Scalarization decisions

After scanning all statements in a function, SRA uses the collected access structures to evaluate whether it can and should create scalar replacements for some or all components of an aggregate. First, we look at all aggregates marked in `should_scalarize_away_bitmap` and not marked in `cannot_scalarize_`

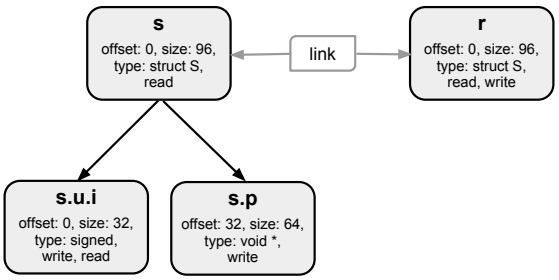

Figure 9: SRA trees of access representatives for example in figure 8.

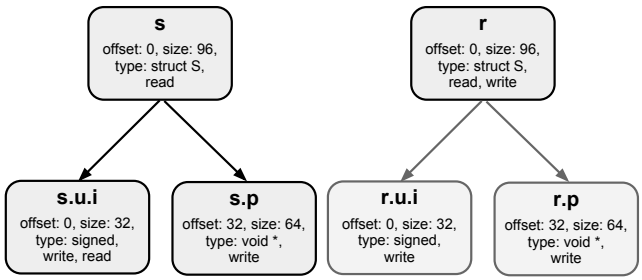

Figure 10: SRA access trees after propagation through assignment links.

`away_bitmap` and if they are simple structures (see goal vii), we create an artificial record for every scalar field in it.

Second, the vectors of pointers to accesses of each candidate are sorted according to their increasing offsets, decreasing sizes and a number of other criteria which put the scalar and most precise types first and stabilize the sort. Afterwards, the vectors are *spliced*. This means that properties of accesses having the same offset and size (they are adjacent after the sort) are aggregated and stored in the first one and these *representatives* of a particular piece of a candidate are connected in a linked list.

The next step is to make this list of individual representatives a list of trees of representatives. In each such tree, a parent represents a part of an aggregate which contains the parts represented by its children. Children are called *subaccesses* of their parents. Such trees cannot be built if there are partial overlaps and in those cases the whole aggregate is disqualified. After building representative trees for all candidates, we process all link structures we marked aggregate assignments with. For each link we create artificial counterparts of subaccesses of the right hand side under the representative of the left hand side if it is possible without introducing partial overlaps (this facilitates goal vi). Moreover, if the same area is represented by a scalar type access on the right hand side and an aggregate one on the left hand side, we change the type of the latter to match the type of the former.

At this point we have all the information to make decisions about what parts of which aggregates are to be replaced with a standalone scalar variable. We traverse the trees for all candidates, ignoring accesses that are marked as unscalarizable (array accesses with variable indices) and scalar ones with subaccesses (to avoid the situation described in goal xi) and create a scalar replacement for every scalar representative if any of the following conditions is true:

- It has been created by total scalarization. The intention is to remove the aggregate altogether.
- It is read on its own more than once. We might save ourselves some loads.
- It is been written to directly and read directly. This might present an opportunity for various scalar propagation passes.
- It is written to directly and some if its ancestors in the access tree is read in an assignment statement.
- It is read directly and some of its ancestors in the access tree is written to in an assignment statement.

These rules do avoid creating scalar replacements in cases like those in figure 7. On the other hand the two last rules help us to do scalarization in both examples in figure 11 which are beneficial because they allow us to remove the variable `s` and associated aggregate assignments entirely (PR 43846). The type of the new replacements is the type of the first access after sorting, therefore the sorting criteria put scalar types with biggest precision first.

Finally, the code traversing the access trees also propagates various information in both directions. Most importantly, we can subsequently always determine whether the data stored in the aggregate part are actually used by examining the `grp_read` flag or whether there are any data which are not scalarized by looking at the `grp_unscalarized_data` flag.

```
struct S
{
  int i;
  int j;
  char c[32];
};
```

```
void bar2(void)
{
  struct S s1l
  s1.i = 6;
  *g = s1;
}

int bar3 (void)
{
   struct S s2;
   s2 = *g;
   return s2.i;
}
```

Figure 11: Simple examples of different treatment of assignments when judging SRA profitability.

3.4 Transformation

Transformation phase again traverses all statements in all basic blocks, examines each one independently and if it needs to be modified, it does so. Modification of references accessing the selected scalars are simple, the memory references are replaced with a lazily instantiated new variable. Aggregate memory references are handled differently for assignments and other statements as described when discussing goal iii and iv (see figures 3 and 4). When deciding whether we can delete a an unnecessary load, it is sufficient to examine the access flags `grp_read` of the left hand side and `grp_unscalarized_data` of the right hand side. When creating complex or vector replacements, we can make them gimple registers only if the flag marking partial writes is not set. The last transformation step is to insert initialization of all scalar replacements of function parameters at the beginning of the function.

3.5 Type incompatibility issues

Unions, typecasts and structures with just one scalar field can lead to situations when a replacement would have different type from the original reference. This can be easily fixed by adding a `VIEW_CONVERT_EXPR` on the right hand side in assignments and by channeling the data transfer through the original aggregate in other statements – this usually happens when a structure with one field is itself replaced with a single aggregate.

A bigger problem might arise when dealing with assignments of aggregates containing unions in GCC 4.5.

```
struct S
{
  unsigned i;
  unsigned j;
};
struct T
{
  char c[64];
};
```

```
union U
{
  struct S s;
  struct T t;
};

void bar4(void)
{
  union U u;
  u.t = g;
  /* some calculations
      with u.s.i
      and u.s.j */
}
```

Figure 12: Example of a problematic union assignment.

Look at the example in figure 12 and assume that SRA decides to create replacements for `u.s.i` and `u.s.j`. Upon encountering the first assignment, the transformation would attempt to load the two new variables from `g.u.i` and `g.u.j` which do not exist. Therefore GCC 4.5 checks that all instantiated replacements can be located on the other side of an assignment and if some cannot, it processes the left and right hand sides separately. Similar problems can arise when propagating subaccesses across assignments and so we have to check that each such access can be located within its base.

GCC 4.6 does not have this problem because the memory references it produces are `MEM_REF`s which can always be constructed regardless of the static type of the base. It still transforms the two sides separately if there are type casts in the original statement due to gimple grammar validation in Ada. We plan to evaluate options for removing that constraint in GCC 4.7.

4 Future work

Apart from trying to remove further constraints from SRA, we are currently considering two other improvements to how GCC handles aggregates:

- perform SRA along hot paths when possible even if it cannot be done in the whole function due to aliasing or partial overlaps, and
- perform at least some simple variant of a real copy propagation on aggregates too because there are cases when total scalarization cannot help. Such propagation would be able to deal with arrays and unions and situations like the one in figure 13.

```
void bar4(struct S s)
{
  struct S t;

  t = s;
  bazz (t);
}
```

Figure 13: Example of a need for aggregate copy propagation.

5 Conclusion

We have shown that SRA has a number of objectives beyond simple replacements of scalar uses and gave bugzilla PRs that show that people do expect the pass to achieve them. We have then described how the pass is organized and what data structure it uses, what they represent and described at least some of the numerous flags in the `access` structure. We have shown that the heuristics which drives scalarization decisions is rather simple once the access trees are built and that it is capable of recognizing the cases when scalarization obviously offers no benefits. We have then briefly described the transformation and the most tricky part, issues with incompatible types.

We have already said that the number of filed bugs against the new SRA implementation was rather surprisingly high. Nevertheless, after we have addressed them the pass seems to work well (save the transition to MEM-REF which required some adjustments) and deliver the expected results.

References

[1] Martin Jambor. Interprocedural optimizations of function parameters. In *Proceedings of the GCC Developers' Summit*, pages 57–63, Montreal, Quebec, Canada, 2009.

[2] Robert Morgan. *Building an optimizing compiler*. Digital Press, Newton, MA, USA, 1998.

[3] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.