Memory SSA - A Unified Approach for Sparsely Representing Memory Operations

Diego Novillo
Red Hat Canada[1]
[1]now at Google Canada

1 Introduction

Static Single Assignment (SSA) provides a sparse data-flow representation for scalars, but it is not well suited for representing objects of aggregate types like structures and arrays, or other memory operations like pointer dereferences and global storage. Several approaches have been proposed to incorporate memory operations into SSA [2, 4, 10, 13, 14]. In general, these approaches build a separate web structure of use-def or def-use chains that are factored very similarly to the basic SSA form.

This paper presents a unified approach for representing both scalars and arbitrary memory expressions in SSA form. It extends the idea of *assignment factoring* [13], but instead of being a separate data structure on top of the CFG, it integrates both scalar and memory SSA forms. The representation maintains all the SSA properties, provides factored use-def chains, its sparseness can be incrementally controlled, it efficiently supports scalarization of memory objects and can use site-specific aliasing information to weed out false positives computed by alias analysis.

2 Virtual SSA Form in GCC

Because of the general undecidability of aliasing [7, 9], memory LOAD/STORE operations may involve many different objects. In general, increasing analysis precision leads to a sparser SSA web, thus freeing up the optimizers to perform more aggressive transformation. However, this usually means larger data sets for the compiler to process, which translates into increased compile times and memory consumption.

GCC uses an approach similar to that used by the SGI compiler [2]. Special compiler generated symbols, called *tags* are created to represent regions of memory and two new *virtual operators* `VDEF`, and `VUSE` are associated with statements that make LOAD/STORE operations.

2.1 Alias Representation

GCC represents memory operations explicitly in the intermediate representation using *virtual operands* [11]. In this scheme, alias and structural analysis creates symbolic names for regions of memory that can be treated as separate objects. When a statement makes a LOAD or STORE operation, GCC adds a virtual operand for every symbolic name associated with that memory location.

For example, given the code fragment in Figure 1(a), GCC creates two symbolic names for the memory region pointed-to by p_3:

The first symbol, called *symbol memory tag* (SMT) is the result of flow-insensitive alias analysis, it (roughly) represents all the memory locations of type `int`. This analysis is almost purely type-based, GCC will try to do a few pruning decisions based on other attributes, but SMTs basically represent memory of a given type. Thus when a pointer of type `int *` is dereferenced in the fragment below, GCC will consider that a reference to symbol `SMT.7`.

The second symbol, called *name memory tag* (NMT), is the result of points-to analysis applied to pointers after the program is in SSA form [12, 6]. Since the analysis is done over SSA names, the results inherit the flow-sensitive properties of the SSA form. Every SSA pointer p_i that has been dereferenced and is found to point to a common set of symbols, is given the same name tag.

Notice that the virtual operands do not actually reference the memory tags. Rather, GCC looks up the alias set associated with each tag (i.e., the set of symbols potentially accessed through that pointer) and adds a virtual operand for each of them. In this case, the memory expression $*p_3$ is a reference to the name tag *NMT*.8 and since *NMT*.8 has symbols *a* and *b* in its may-alias set, the compiler inserts two virtual operands, one for *a* and another for *b*.

One may wonder why did GCC bother creating a type tag (*SMT*.7) in this case. Type tags are used when the compiler cannot find a name tag associated to a particular SSA pointer. In those cases, the compiler retrieves the type tag associated with the SSA name's base symbol (*p* in this case). If the base symbol does not have an associated type tag, the compiler will abort. SSA pointers may "lose" their name tags in various ways: when points-to analysis could not compute a known set of pointed-to symbols, or when other transformations create new SSA pointers and fail to compute a name tag for them. Note that these are not necessarily error conditions, nor do they indicate loss of precision. It may happen that all the SSA names for a pointer **do** point to the same set of symbols that would be computed by the flow-insensitive analysis. In which case, using the type tag is perfectly reasonable.

2.2 Representation of aggregates

Aliasing is not the only situation where the compiler creates artificial symbols to represent regions of memory. References to aggregate types like structures and arrays are also handled in a similar fashion. Whenever possible, the compiler will create symbolic names to represent distinct regions inside aggregates (called *structure field tags* or SFT). For instance, in Figure 2(b), GCC will create three SFT symbols for this structure, namely `SFT.0` for `A.x`, `SFT.1` for `A.b` and `SFT.2` for `A.a`.

Once the artificial `SFT` symbols have been computed, they are added as virtual operands to the appropriate statements and those virtual operands are then put into SSA form. Furthermore, this approach to assigning symbolic names to regions of aggregates can be combined with alias analysis, so the `SFT` symbols may be added to may-alias sets, if necessary.

3 Problems with this approach

The compiler adds a virtual operand for each member of the may-alias set for the sym-

```
foo (int i)
{
  if (i_2 > 10)
    t.0_18 = &a
  else
    t.0_17 = &b

  # t.0_1 = PHI <t.0_18(3), t.0_17(4)>
  p_3 = t.0_1
  D.1527_4 = i_2 + 1



  *p_3 = D.1527_4
  D.1528_5 = i_2 + 2

  #   a_7 = VDEF <a_6>
  a = D.1528_5
  D.1529_8 = i_2 + 10

  #   b_10 = VDEF <b_9>
  b = D.1529_8

  #   VUSE <a_7>
  a.1_11 = a

  #   VUSE <b_10>
  b.2_12 = b
  D.1533_13 = a.1_11 + b.2_12



  D.1534_14 = *p_3
  D.1530_15 = D.1533_13 + D.1534_14

  return D.1530_15
}
```

```
p, int *, type memory tag: SMT.7,
   may aliases: { a b }
p_3, name memory tag: NMT.8,
   may aliases: { a b }

foo (int i)
{
  if (i_2 > 10)
    t.0_18 = &a
  else
    t.0_17 = &b

  # t.0_1 = PHI <t.0_18, t.0_17>
  p_3 = t.0_1
  D.1527_4 = i_2 + 1

  #   a_19 = VDEF <a_6>
  #   b_20 = VDEF <b_9>
  *p_3 = D.1527_4
  D.1528_5 = i_2 + 2

  #   a_7 = VDEF <a_19>
  a = D.1528_5
  D.1529_8 = i_2 + 10

  #   b_10 = VDEF <b_20>
  b = D.1529_8

  #   VUSE <a_7>
  a.1_11 = a

  #   VUSE <b_10>
  b.2_12 = b
  D.1533_13 = a.1_11 + b.2_12

  #   VUSE <a_7>
  #   VUSE <b_10>
  D.1534_14 = *p_3
  D.1530_15 = D.1533_13 + D.1534_14

  return D.1530_15
}
```

Figure 1: Current representation of memory operations.

```
struct A
{
  int a
  int b
  float x
}

foo ()
{

  a.a = 30


  a.b = 10


  D.1527 = a.a
  D.1528 = (float) D.1527
  D.1529 = D.1528 * 2.5e+0


  a.x = D.1529


  D.1531 = a.x


  D.1532 = a.b
  D.1533 = (float) D.1532
  D.1530 = D.1531 + D.1533

  return D.1530
}
```

(a) Original code before SFT creation.

```
struct A
{
  int a
  int b
  float x
}

foo ()
{
  #   SFT.2_2 = VDEF <SFT.2_1>
  a.a = 30

  #   SFT.1_4 = VDEF <SFT.1_3>
  a.b = 10

  #   VUSE <SFT.2_2>
  D.1527_5 = a.a
  D.1528_6 = (float) D.1527_5
  D.1529_7 = D.1528_6 * 2.5e+0

  #   SFT.0_9 = VDEF <SFT.0_8>
  a.x = D.1529_7

  #   VUSE <SFT.0_9>
  D.1531_10 = a.x

  #   VUSE <SFT.1_4>
  D.1532_11 = a.b
  D.1533_12 = (float) D.1532_11
  D.1530_13 = D.1531_10 + D.1533_12

  return D.1530_13
}
```

(b) SFTs in SSA form.

Figure 2: Current representation of memory operations.

bol associated with a memory expression. Since alias analysis results are often conservative, may-alias sets may contain tens and even hundreds of symbols (about 600 in the collection of GCC, MICO, POOMA, SPEC2000 and TRAMP3D). In some extreme cases, an alias set may contain millions of elements. In FreeFEM3D (`http://www.freefem.org/`), the generated parser file `parse.ff.cc` produces alias sets with 4.8 million elements. Therefore, every memory operation will have as many virtual operands as the may-alias set.

While the current approach succeeds in keeping the different members of an alias set independent of each other, it penalizes memory expressions involving all the alias sets (i.e., LOAD and STORE operations via the associated memory tag).

To alleviate this problem, GCC includes a grouping heuristic similar to that used in the SGI compiler [2]. All the members of the same alias set are grouped under the same symbol name. This drastically reduces the number of virtual operands, but also increases the density of use-def chains, thus preventing many optimization opportunities. For instance, given the expression `p = (i > 10) ? &a : (i > 20) ? &b : &c`, the may-alias set for `p`'s memory tag is `{a, b, c}`. Therefore, LOAD and STORE operations via `*p` will contain virtual operands for each of `a`, `b` and `c`. References to each member of the alias set will not affect the others. This gives freedom to the optimizer to do transformations that would not be possible otherwise (e.g., in Figure 3 propagate the value 5 from line 15 associated with a_6 into line 26).

The grouping heuristic used by GCC will associate all the members of the same alias set with the same, resulting in the SSA form in Figure 3(b). Notice that while the representation size has been greatly reduced, the use-def chains are now linking the load of `a` at line 26 with the store to `b` at line 18. Which blocks the store at line 15 to be propagated into the load at line 26.

4 Memory SSA

The approach proposed in this paper attempts to preserve the sparse properties of the current virtual SSA form implemented by GCC, while avoiding the excessive overhead resulting from voluminous alias sets. The design of this new approach has been driven by the following desirable properties:

Integration. While scalar SSA and memory factored SSA have different properties, they both use the same basic versioning scheme and provide a unified view of the program to the optimizers. This facilitates implementation details, as there is no need for the optimizers to consult on-the-side data structures to determine if a transformation is valid.

Sparseness. As much as possible, LOAD and STORE operations to memory locations that may not overlap should not affect each other.

Lightweight. In general, this is inversely proportional to sparseness. The sparser the representation is, the higher the number of use-def chains or SSA versions that need to be kept around. In the case of the current implementation in GCC, this also implies a large increase in the number of virtual operands associated with LOAD and STORE operations.

Stability. In some cases, a side effect of optimization results in a reduction of memory operands. Consider the program in Figure 4(a), variable `a` has its address taken

```
1  foo (i)
2  {
3    if (i_3 > 10)
4      tmp_17 = &a;
5    else
6      if (i_3 > 20)
7        tmp_16 = &b;
8      else
9        tmp_15 = &c;
10
11   # tmp_2 = PHI <tmp_16, tmp_15, tmp_17>;
12   p_4 = tmp_2;
13
14   #   a_6 = VDEF <a_5>;
15   a = 5;
16
17   #   b_8 = VDEF <b_7>;
18   b = 3;
19
20   #   VUSE <a_6>;
21   #   VUSE <b_8>;
22   #   VUSE <c_18>;
23   D.1530_9 = *p_4;
24
25   #   VUSE <a_6>;
26   a.2_10 = a;
27
28   D.1532_11 = D.1530_9 + a.2_10;
29   D.1529_12 = (float) D.1532_11;
30   return D.1529_12;
31 }
```

(a) Ungrouped alias sets. Alias set members are independent of each other.

```
1  foo (i)
2  {
3    if (i_3 > 10)
4      tmp_17 = &a;
5    else
6      if (i_3 > 20)
7        tmp_16 = &b;
8      else
9        tmp_15 = &c;
10
11   # tmp_2 = PHI <tmp_16, tmp_15, tmp_17>;
12   p_4 = tmp_2;
13
14   #   TMT.7_19 = VDEF <TMT.7_18>;
15   a = 5;
16
17   #   TMT.7_20 = VDEF <TMT.7_19>;
18   b = 3;
19
20
21
22   #   VUSE <TMT.7_20>;
23   D.1530_9 = *p_4;
24
25   #   VUSE <TMT.7_20>;
26   a.2_10 = a;
27
28   D.1532_11 = D.1530_9 + a.2_10;
29   D.1529_12 = (float) D.1532_11;
30   return D.1529_12;
31 }
```

(b) Grouped alias sets. Alias set members affect each other.

Figure 3: Difference between grouped and ungrouped alias sets.

by pointer p_1, therefore references to *p are represented in virtual SSA form as *may* references to a. However, constant propagation will prove that p will only ever point to a by propagating &a into all dereferences of p[1]. This means that a can now be converted into scalar SSA form (Figure 4(b)). This transition between virtual and scalar SSA form can be costly, depending on how much work needs to be done to transition from one form to the other.

The basic premise in Memory SSA is similar to the current virtual SSA form as implemented in GCC. Every memory STORE operation represents a *potential* definition for the objects that may be residing at that location. This is usually referred to as a *preserving* or *non-killing* definition in the literature. If a STORE operation may affect more than one memory object, we refer to it as a *factored store*. Similarly, memory LOAD operations represent *potential* reads for the objects associated with that memory location. If a LOAD operation may read from more than one memory object, we refer to it as a *factored load*.

Memory SSA attacks the memory consumption problem by systematically reducing the number of SSA names created at every factored store. This reduction works using a grouping mechanism similar to the one described earlier, but instead of being an "all or none" grouping, it can be gradually adjusted to allow a balance between alias precision information and memory consumption.

The grouping done by Memory SSA is based on the concept of memory partitions. Each memory partition represents an arbitrary set of memory symbols, so that if symbol V belongs to memory partition P, every reference to V will be converted into a reference to partition P. Since the number of partitions is a fraction of the total number of symbols, this approach reduces the amount of virtual operands, SSA names and PHI nodes needed. In turn, this reduces compilation times because there is simply fewer things for the optimizers to deal with.

The drawback of grouping schemes is that it may reduce the representation precision as discussed in Section 3. Note that in the context of this document, we are not interested in the precision obtained by alias analysis. Rather, we are interested in the precision used to **represent** the alias sets computed by alias analysis.

Given a memory reference M, alias analysis determines that M affects a set of symbols $S1$. The representation of that memory reference is a set of virtual operands $S2$ such that $S2 \supseteq S1$. We say that this representation is *precise* if $S2 = S1$ for every memory reference in the program. Otherwise, the representation is *imprecise*.

Imprecise representations are always safe from the point of view of correctness because they represent more memory symbols than necessary. However, they tend to block optimization opportunities as shown in Figure 3(b). The store $a = 5$ at line 15 cannot be propagated to the load of a in line 26 because of the unrelated store to b at line 18.

We distinguish two types of partitioning schemes, namely dynamic and static. Dynamic partitioning is decided during the SSA renaming process. Every memory store generates exactly one SSA name which is associated with the set of symbols stored by the statement. Static partition, on the other hand, is determined before renaming. All the symbols in a single partition set are represented by an artificial symbol called Memory Partition Tag

[1] This example is for illustration only. This is not exactly how the current GCC implementation works, but the final effect is the same.

```
foo ()
{
  p_1 = &a;

  #   a_6 = VDEF <a_5>;
  *p_1 = 3;

  #   VUSE <a_6>;
  D.1525_2 = *p_1;

  D.1524_3 = D.1525_2 + 5;
  return D.1524_3;
}
```

(a) SSA form before constant propagation.

```
foo ()
{
  p_1 = &a;


  a_7 = 3;


  D.1525_2 = a_7;

  D.1524_3 = D.1525_2 + 5;
  return D.1524_3;
}
```

(b) SSA form after constant propagation.

Figure 4: Transition from virtual to scalar SSA form for aliased variables.

(MPT). Both techniques are described in the following sections.

4.1 Dynamic memory partitions

Every factored store generates exactly one SSA name for an artificial symbol called `MEM`. Initially, every aliased memory expression refers to the same SSA name $\texttt{MEM}_1$. `STORE` operations create a new version of `MEM` and associate that version to all the objects related to that particular memory expression. Similarly, `LOAD` operations receive their value from a set of reaching `MEM` versions, associated to each of the objects related to the underlying memory expression.

`STORE` operations generate exactly one SSA name for `MEM` and serve as the merge point for all the SSA names that reach the objects associated with that memory location. This is represented with the `VDEF` operator:

$$\texttt{MEM}_i = \texttt{VDEF}\ \ <\texttt{MEM}_x, \texttt{MEM}_y, \texttt{MEM}_z>.$$

which indicates that SSA name $\texttt{MEM}_i$ is generated by "*killing*" the reaching SSA versions x, y and z.

`LOAD` operations are represented with the `VUSE` operator, which takes as operands all the reaching definitions for the objects associated with the memory location.

The SSA numbering for the `MEM` object is done using a slight modification of the standard SSA renaming algorithm [3].

- Every `store` operation is associated with a *memory tag* (i.e., one of `SMT`, `NMT` or `SFT`). When the SSA renamer finds a `store` for a memory expression E, it will

  1. Determine what tag T is associated with E. If T is a type or name tag, it will retrieve its associated may-alias set S.
  2. For every aliased symbol V in S,

if E does conflict with V[2], add $currdef(V)$ to the set of *killed names*.

3. Generate a new SSA name $\texttt{MEM}_i$ and set $currdef(V) = \texttt{MEM}_i$.
4. Add the VDEF operator: $\texttt{MEM}_i = \texttt{VDEF} <\{killed\ names\}>$

- Every `load` operation is processed similarly to step 2 above. And a VUSE operator is added as VUSE $<\{reaching\ names\}>$

As an example, consider the code fragment in Figure 5(a). After alias analysis, GCC will currently generate the SSA form in Figure 5(b). For brevity, the pointer setup code that establishes the points-to sets computed for p_5 and q_6 has been elided.

Using the memory SSA renaming outlined before, the SSA form is as shown in Figure 6. There are a few things worth noting in this approach:

1. The same MEM version may be "killed" more than once. After all, these are preserving definitions, so the reaching SSA name is only used to preserve the factored use-def and def-def chains. For instance, stores to a and b at lines 3 and 6 provide a new name for $\texttt{MEM}_7$.
2. STOREs to distinct members of the same alias set are not linked in the same use-def or def-def chain.
3. STOREs to a memory tag (i.e., pointer dereference expression) serve as merge points for all the outstanding MEM names for every member of the tag's alias set. Notice how the name created by the STORE to $*\texttt{p}_5$ at line 14 is then used by the LOAD from b at line 17.

In general, the size of the set of operands on the right hand side of VDEF or VUSE operators will depend on the number of distinct current reaching definitions for members in the tag's alias set.

Call clobbered variables and other global storage (global variables) is modeled similarly. All call clobbered objects are grouped in a single set, the renaming mechanism is exactly the same (Figure 7). Notice how the second call clobbering site at line 19 uses the factored name created by the first clobbering site at line 13.

This new approach is, in principle, more stable with respect to changes to may-alias sets. As illustrated in Figure 4(b), alias sets may change due to optimizations. The current virtual SSA form used by GCC forces the compiler to perform some expensive incremental updates of the SSA form. The new form should, in principle, allow for a much quicker incremental update.

Currently, GCC creates artificial symbols to represent name tags, type tags and structure field tags. This is necessary so that they can be added as operands to the virtual operators VDEF and VUSE. This unnecessarily pollutes the symbol table and would not be necessary under the new scheme. These tags may just be indices or hash values into a lookup data structure containing may-alias sets.

4.2 Static memory partitions

Although dynamic partitions guarantee that factored stores will only create one SSA name, the fact that they create overlapping live ranges has been found to be problematic

[2]May-alias sets may be imprecise (particularly type-based sets). In some cases, it is possible to determine whether a particular memory expression may actually overlap with a variable or not.

```
foo (i)
{
  ...

  a = 2


  b = 5


  b.3 = b

  D.1536 = b.3 + 3




  *p = D.1536


  b.3 = b
  D.1537 = 10 - b.3



  *q = D.1537


  a.4 = a


  X.x = a.4
  return
}
```

(a) Before conversion into SSA form.

```
p5 points-to {a, b, c}
q6 points-to {b, c}

foo (i)
{
  ...
  #   a8 = VDEF <a7>;
  a = 2;

  #   b10 = VDEF <b9>;
  b = 5;

  #   VUSE <b10>;
  b.311 = b;

  D.153612 = b.311 + 3;

  #   a25 = VDEF <a8>;
  #   b26 = VDEF <b10>;
  #   c27 = VDEF <c24>;
  *p5 = D.153612;

  #   VUSE <b26>;
  b.313 = b;
  D.153714 = 10 - b.313;

  #   b28 = VDEF <b26>;
  #   c29 = VDEF <c27>;
  *q6 = D.153714;

  #   VUSE <a25>;
  a.415 = a;

  #   X17 = VDEF <X16>;
  X.x = a.415;
  return;
}
```

(b) After conversion into SSA form.

Figure 5: SSA form with virtual operands as currently implemented in GCC.

p_5 points−to {a, b, c}
q_6 points−to {b, c}

CD(v) means that the generated MEM i name is the "current definition" for v.
LU(v) looks up the "current definition" for v.

The initial SSA name for MEM is MEM_7.

```
foo (i)
{
      1    ...
      2    #   MEM_8 = VDEF <MEM_7>              ⇒ CD(a)
      3    a = 2
      4
      5    #   MEM_10 = VDEF <MEM_7>             ⇒ CD(b)
      6    b = 5
      7
      8    #   VUSE <MEM_10>                     ⇒ LU(b)
      9    b.3_11 = b
     10
     11    D.1536_12 = b.3_11 + 3
     12
     13    #   MEM_25 = VDEF <MEM_8, MEM_10>     ⇒ CD(a, b, c)
     14    *p_5 = D.1536_12
     15
     16    #   VUSE <MEM_25>                     ⇒ LU(b)
     17    b.3_13 = b
     18    D.1537_14 = 10 − b.3_13
     19
     20    #   MEM_26 = VDEF <MEM_25>            ⇒ CD(b, c)
     21    *q_6 = D.1537_14
     22
     23    #   VUSE <MEM_25>                     ⇒ LU(a)
     24    a.4_15 = a
     25
     26    #   MEM_17 = VDEF <MEM_7>             ⇒ CD(SFT.2)
     27    X.x = a.4_15
     28    return
}
```

Figure 6: Memory SSA form for program in 5(a).

Call clobbered symbols { A, B, C, D, x, y, z, L, N }
p points to {a, b}

CD(v) means that the generated MEM i name is the "current definition" for v.
LU(v) looks up the "current definition" for v.

The initial SSA name for MEM is MEM_1.

```
foo (i)
{
      1    # MEM_2 = VDEF <MEM_1>                          ⇒ CD(A)
      2    A = ...
      3
      4    # MEM_3 = VDEF <MEM_1>                          ⇒ CD(B)
      5    B = ...
      6
      7    # MEM_4 = VDEF <MEM_1>                          ⇒ CD(C)
      8    C = ...
      9
     10    # MEM_5 = VDEF <MEM_1>                          ⇒ CD(D)
     11    D = ...
     12
     13    # MEM_6 = VDEF <MEM_1, MEM_2, MEM_3, MEM_4, MEM_5>   ⇒ CD(A, B, C, D, x, y, z, L, N)
     14    bar ();
     15
     16    VUSE <MEM_1>                                   ⇒ LU(a, b)
     17    ... = *p;
     18
     19    # MEM_7 = VDEF <MEM_6>                          ⇒ LU(A, B, C, D, x, y, z, L, N)
     20    baz ();
}
```

Figure 7: Memory SSA form for global storage.

for certain memory optimizations like PRE (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=29680). During implementation, PRE had to be disabled and the presence of overlapping live ranges may be preventing optimizations like store-motion to work as implemented today because unrelated stores may be killing the same SSA name.

Another problem we found with pure dynamic partitions was an explosion in the number of PHI nodes. Recall that every factored store S becomes a definition site for every symbol factored by that store. Therefore, basic blocks at the dominance frontier of S will require a ϕ node for each symbol store by S. This was essentially undoing all the factoring effects because ϕ nodes were separating all the symbols that had been factored by a previous store. This is illustrated in Figure 8.

Notice that all of the ϕ nodes in Figure 8 are not necessary (they all have the exact same arguments). The same information can be obtained with a single ϕ node. Besides the memory consumption problems caused by an excessive number of unnecessary ϕ nodes, they also cause a considerable compile time slowdown because the SSA renamer must compute dominance frontier information for every symbol separately.

These problems can be addressed using static memory partitions. A memory partition tag (MPT) is a symbol that represents a fixed set of aliased or call-clobbered symbols. There does not need to be an alias or type relationship between an MPT and the symbols in its set.

The partitioning can be completely arbitrary, and it can even change as long as this rule is followed: Given two partitions MPT.i and MPT.j, they must not contain symbols in common. From this, it follows that given a symbol V, V may only belong to exactly one partition.

When the operand scanner finds symbol V in a memory expression, it asks whether V belongs to a partition, if it does, then an operand is added for V's partition.

This static partitioning can be used directly or to complement dynamic partitioning.

4.2.1 Pure static partitioning

Once computed, memory partitions can be used as direct replacement of all the grouped memory symbols. This way, instead of dealing with statements producing hundreds of virtual operators, only a handful will be required. This threshold can be configured using the parameters `-param max-aliased-vops` and `-param avg-aliased-vops`

While this representation is not precise, the heuristics used for grouping can be altered to minimize the grouping side-effects. The heuristic counts the total number of memory references in the function. With that, it will estimate the number of virtual operators needed for stores and loads and compare them against two thresholds:

- A maximum number of virtual operators allowed for the whole function (`max-aliased-vops`).
- An average number of virtual operators allowed per statement (`avg-aliased-vops`).

If both values are below the threshold, nothing is done. Otherwise, the heuristic in `compute_memory_partitions` triggers and symbols are added to a work list. Given a memory variable V in the list, its "partitioning score" (*pscore*) is a weighted given by the following formula:

```
foo (i)
{
  ... p_2 is set to point to { a, b, c, d, e, f, g } ...

  if (i > 10)
    # MEM_3 = VDEF <MEM_1>          STORES { a, b, c, d, e, f, g }
    *p_2 = ...
  else
    # MEM_6 = VDEF <MEM_1>          STORES { a, b, c, d, e, f, g }
    *p_2 = ...
  endif
  # a_7 = PHI <MEM_3, MEM_6>
  # b_8 = PHI <MEM_3, MEM_6>
  # c_9 = PHI <MEM_3, MEM_6>
  # d_10 = PHI <MEM_3, MEM_6>
  # e_11 = PHI <MEM_3, MEM_6>
  # f_12 = PHI <MEM_3, MEM_6>
  # g_13 = PHI <MEM_3, MEM_6>

  # VUSE <a_7, b_8, c_9, d_10, e_11, f_12, g_13>
  x_15 = *p_2
}
```

Figure 8: Unfactoring effects due to ϕ nodes and dynamic partitions.

$$frequency\_writes * 64 + frequency\_reads * 32 + num\_direct\_writes * 16 + num\_direct\_reads * 8 + num\_indirect\_writes * 4 + num\_indirect\_reads * 2 + noalias\_state$$

Where

frequency_writes is the aggregate execution frequency of all the write operations to V.

frequency_reads is the aggregate execution frequency of all the read operations from V.

num_direct_writes is the number of direct write operations to V.

num_direct_reads is the number of direct read operations from V.

num_indirect_writes is the number of indirect (i.e., through a pointer or call site) write operations to V.

num_indirect_reads is the number of indirect read operations from V.

noalias_state is an integer in the range $0-4$ indicating the value of the family of flags `-fargument-noalias-*`.

The higher this score is for V, the less likely that V will be added to a partition.

This makes the partitioning better, particularly in small functions (where we just don't care about how many virtual operators are needed) and allows a much smoother control over the partitioning behaviour.

There are 3 different preset values for the parameters `max-aliased-vops` and `avg-aliased-vops`. One for each of -O1, -O2 and -O3. At -O1 the idea is to make compilation time very quick. The current values give

us about 1-2% memory savings at -O1 and a 3-5% compile-time savings.

For -O2, the compile-time and memory utilization is roughly the same (though in some cases, you'll notice an increase, so I may have to adjust further).

At -O3, the settings should be such that we very rarely partition.

4.2.2 Hybrid partitioning

As discussed earlier, pure dynamic partitioning leads to a proliferation of ϕ nodes that essentially undo all the benefits of factored stores. To address this problem, we can use the static partitions when placing ϕ nodes. Instead of computing dominance frontiers and creating ϕ nodes for every individual memory symbol, the compiler places ϕ nodes using partitions. This reduces precision as ϕ nodes will now group symbols that may have had unrelated stores.

For instance, suppose that x and y belong to the same partition $MPT.1$. The load of y at line 11 is now reached by the unrelated store to x in line 3.

```
1  if (...)
2    # x_4 = VDEF <x_3>
3    x = ...
4  else
5    # y_5 = VDEF <y_2>
6    y = ...
7  endif
8  # MPT.1_9 = PHI <x_4, y_5>
9
10 # VUSE <MPT.1_9>
11 k_3 = y
```

Another effect that occurs when mixing static and dynamic partitions is that it is now possible for a ϕ node to have multiple reaching definitions for a **single** argument. Recall that with dynamic partitioning, every single store generates a name for the unique symbol being stored. Only factored stores will generate a name for `MEM`. In contrast, with static partitioning individual stores to grouped symbols are considered stores to their holding partition.

For instance, using static partitioning, stores to x and y will always affect each other:

```
1  if (...)
2    # MPT.1_3 = VDEF <MPT.1_1>
3    x = ...
4
5    # MPT.1_4 = VDEF <MPT.1_3>
6    y = ...
7  else
8    # MPT.1_5 = VDEF <MPT.1_1>
9    x = ...
10 endif
11 # MPT.1_6 = PHI <MPT.1_4, MPT.1_5>
```

Notice how the unrelated stores to x and y in lines 3 and 6 are linked with a def-def link. But using dynamic partitioning, this link is not needed:

```
1  if (...)
2    # x_3 = VDEF <x_1>
3    x = ...
4
5    # y_4 = VDEF <y_2>
6    y = ...
7  else
8    # x_5 = VDEF <x_1>
9    x = ...
10 endif
11 # MPT.1_6 = PHI <???, x_5>
```

By splitting the stores to x and y we avoid unnecessary def-def links but this creates a secondary problem: the first argument for the ϕ node at line 11 needs to be reached by **two** different names, namely x_3 and y_4.

The first approach I tried was to split these ϕ nodes during renaming so that instead of having

a single ϕ node for MPT.1, the renamer would create one ϕ node for x and another for y. This approach proved to be extremely hard to implement and brittle.

Since new ϕ nodes would appear **during** renaming, this meant that the renamer would frequently need to go back to rename dominance sub-trees that had already been renamed because ϕ nodes higher up in the dominance hierarchy had been split **after** the children sub-trees had been renamed. This also slowed down the renaming process and greatly increased implementation complexity.

An alternate approach involves a new factoring device to gather the multiple reaching definitions in these cases. This new device, called σ node is inserted during renaming when necessary. A σ node acts as a "sink" that receives all the conflicting reaching definitions and produces a new name that can be used as the argument for the receiving ϕ node. In the previous example, we would have

```
if (...)
  # x3 = VDEF <x1>
  x = ...

  # y4 = VDEF <y2>
  y = ...

  MPT.16 = SIGMA <x3, y4>
else
  # x5 = VDEF <x1>
  x = ...
endif
# MPT.17 = PHI <MPT.16 x5>
```

Note that though this mechanism is fairly straightforward and does not require complex changes in the renamer, it is still very experimental. It is not yet clear whether it provides major benefits in code generation versus a pure static approach.

5 Experimental results

The current implementation uses pure static partitioning. The hybrid partitioning scheme is being implemented on the `mem-ssa` branch.

Experiments use a set of C and C++ files taken from various applications: GCC, SPEC 2000, MICO (a Corba implementation), DLV (a disjunctive catalog system), TraMP-3d (astrophysical hydrodynamics simulation) and a few test cases from GCC's bugzilla database.

Static partitioning shows marked improvements in compile times, on average compilation times are 5% to 25% faster than before. The following tables summarize the compile time speedups obtained by the Memory SSA implementation. Phase timing is reported by GCC using the switch `-ftime-report`. Only the phases that showed significant changes are included. All the timings were measured on an Intel Core Duo 64bits @2.13 Ghz. Figures 9, 10, 11, 12, 13 and 14 show the compile time improvements obtained with static partitioning on GCC, DLV, MICO, SPEC 2000, TraMP-3D and PR 12850 (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=12850`) respectively.

6 Related Work

SGI's approach is to use as many different symbols as possible [2], but they still link all the names for the same symbol. All the variables that are grouped under the same symbol are always linked together. Use-def chains over these symbols are sparse, but every use is linked to every definition.

Steensgaard proposed assignment factoring [13]. It overlays factored use-def links over

Phase	Before	After	% change
tree PTA	16.39	15.35	-6.3%
tree alias analysis	12.33	11.75	-4.7%
tree SSA rewrite	6.24	5.13	-17.8%
tree SSA incremental	15.79	12.09	-23.4%
tree operand scan	85.90	52.28	-39.1%
TOTAL	476.43	437.47	-8.2%

Figure 9: Compile time improvements on GCC

Phase	Before	After	% change
tree PTA	4.22	3.69	-12.6%
tree SSA incremental	4.09	2.89	-29.3%
tree operand scan	20.85	14.47	-30.6%
TOTAL	101.39	91.69	-9.6%

Figure 10: Compile time improvements on DLV

Phase	Before	After	% change
tree SSA rewrite	4.56	3.13	-31.4%
tree SSA incremental	10.00	5.93	-40.7%
tree operand scan	90.19	50.24	-44.3%
TOTAL	444.08	391.64	-11.8%

Figure 11: Compile time improvements on MICO

Phase	Before	After	% change
tree alias analysis	6.01	5.31	-11.6%
tree SSA rewrite	4.54	2.66	-41.4%
tree SSA incremental	9.47	5.08	-46.4%
tree operand scan	49.93	43.83	-12.2%
TOTAL	287.97	271.64	-5.7%

Figure 12: Compile time improvements on SPEC 2000

Phase	Before	After	% change
tree SSA rewrite	3.80	2.71	-28.7%
tree SSA incremental	3.29	1.92	-41.6%
tree operand scan	22.72	16.87	-25.7%
expand	5.88	5.06	-13.9%
TOTAL	88.78	77.68	-12.5%

Figure 13: Compile time improvements on TraMP-3D

Phase	Before	After	% change
tree alias analysis	4.61	3.66	-20.6%
tree SSA rewrite	4.77	3.10	-35.0%
tree SSA incremental	5.85	1.87	-68.0%
tree operand scan	54.29	26.98	-50.3%
TOTAL	159.39	118.16	-25.9%

Figure 14: Compile time improvements on PR12850

the CFG. No details on application to aggregate types or global storage. Store fragmentation (i.e. points-to analysis) determines what statements conflict. No details on forming the SSA form nor keeping it up-to-date.

Cytron et.al. proposed iterating SSA formation using results of alias analysis [4]. But the approach is too expensive, it requires building the SSA form over and over.

Choi et.al. proposed location factoring [1]. This is a precursor to Steensgaard's assignment factoring where use-def links are inserted between statements that access the same memory location. It's the same idea, but less factored than assignment factoring.

Fink et.al. proposed Heap SSA [5]. Based on Array SSA [8].

References

[1] J.-D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, February 1994.

[2] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *Computational Complexity*, pages 253–267, 1996.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in SSA form. *ACM SIGPLAN Notices*, 28(6):36–45, 1993.

[5] S. J. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object references in strongly typed languages. In *Static Analysis Symposium*, pages 155–174, 2000.

[6] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[7] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[8] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Symposium on Principles of Programming Languages*, pages 107–120, 1998.

[9] William Landi. Undecidability of static analysis. *ACM Letters on Programming*

Languages and Systems, 1(4):323–337, December 1992.

[10] C. Lapkowski and L. J. Hendren. Extended SSA Numbering: Introducing SSA Properties to Language with Multi-level Pointers. In *Computational Complexity*, pages 128–143, 1998.

[11] D. Novillo. Design and Implementation of Tree SSA. In *2004 GCC Developers' Summit*, pages 119–130, 2004.

[12] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of the ACM workshop on Program Analysis for Software Tools and Engineering*. ACM Press, 2004.

[13] B. Steensgaard. Sparse functional stores for imperative programs. *ACM SIGPLAN Notices*, 30(3):62–70, 1995.

[14] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.