Alias Analysis in GCC

Diego Novillo

dnovillo@redhat.com

Red Hat Canada

GCC Moscow Meeting

Moscow, Russia, August 2006

Introduction

- Two levels of alias analysis
- GIMPLE
  - Points-to analysis (field/flow sensitive)
  - Type-based analysis used as fallback
  - Explicitly represented in IL using memory tags
- RTL
  - Query based built on type aliases
  - Pairwise disambiguation system

Memory expressions in GIMPLE

- Memory variables → `!is_gimple_reg(v)`
  - Aggregate types → `AGGREGATE_TYPE_P`
  - `needs_to_live_in_memory`
    - Globals → `is_global_var`
    - Addressables → `TREE_ADDRESSABLE`
    - Return values of an aggregate type
  - Volatiles, hard registers, non-promoted `complex`.
- GIMPLE memory may end up in registers

`!is_gimple_reg` `needs_to_live_in_memory`

Memory expressions in GIMPLE

- At most one memory load and one memory store per statement
  - Loads only allowed on RHS of assignments
  - Stores only allowed on LHS of assignments
- Gimplifier will enforce this property
- Dataflow on memory represented explictly
  - Factored Use-Def (FUD) chains or “Virtual SSA”
  - Requires a symbolic representation of memory

Symbolic Representation of Memory

- Since we want an SSA-like representation, we need symbols to represent memory
- Unaliased memory
  - Globals → base symbol
  - Addressables → base symbol
  - Aggregates → base symbol or field tags (SFT)
- Aliased memory
  - Dereferences → memory tags (SMT/NMT)

Symbolic Representation of Memory

- Aliased memory referenced via pointers
- GIMPLE only allows single-level pointers

Invalid	Valid
`**p`	`t.1 = *p` `*t.1`
`*(a[3].ptr)`	`t.1 = a[3].ptr` `*t.1`

Symbolic Representation of Memory

- Pointer `P` is associated with <u>memory tag</u> `MT`
  - `MT` represents the set of variables pointed-to by `P`
- So `*P` is a reference to `MT`

```
if (...)
  p = &a
else
  p = &b
*p = 5
```

p points-to `{a, b}`
p has memory tag `MT`

Interpreted as `MT = 5`

Associating Memory with Symbols

- Alias analysis
  - Builds points-to sets and memory tags
- Structural analysis
  - Builds field tags (aka sub-variables)
- Operand scanner
  - Scans memory expressions to extract tags
  - Prunes alias sets based on expression structure

Alias Analysis

- Points-to alias analysis (PTAA)
  - Based on constraint graphs
  - Field and flow sensitive, context insensitive
  - Intra-procedural (inter-procedural in 4.2)
  - Fairly precise
- Type-based analysis (TBAA)
  - Based on input language rules
  - Field sensitive, flow insensitive
  - Very imprecise

Alias Analysis

- Two kinds of pointers are considered
  - Symbols: Points-to is flow-insensitive
    - Associated to Symbol Memory Tags (SMT)
  - SSA names: Points-to is flow-sensitive
    - Associated to Name Memory Tags (NMT)
- Given pointer dereference `*ptr`$_{42}$
  - If `ptr`$_{42}$ has NMT, use it
  - If not, fall back to SMT associated with `ptr`

Alias Analysis

- After alias analysis
  - Every dereferenced symbol pointer will have an associated SMT
  - Most dereferenced SSA pointers will have an associated NMT
  - Variables whose address escapes local function are considered call-clobbered → important when processing CALL_EXPRs

Structural Analysis

- Separate structure fields are assigned distinct symbols

```
struct A
{
  int x;
  int y;
  int z;
};

struct A a;
```

- Variable a will have 3 sub-variables
  `{ SFT.1, SFT.2, SFT.3 }`

- References to each field are mapped to the corresponding sub-variable

IL Representation

- Memory tags need to be represented but original expressions cannot be rewritten
- GCC's approach: virtual operators
  - `V = V_MAY_DEF <V>`
    - Symbol V is partially or potentially stored by stmt
  - `VUSE <V>`
    - Symbol V is partially or potentially loaded by stmt
  - `V = V_MUST_DEF <V>` ← **deprecated**
    - Symbol V is totally and definitely clobbered by stmt

IL Representation

```
foo (i, a, b, *p)
{
  p =(i > 10) ? &a : &b
  *p = 3
  return a + b
}
```

```
foo (i, a, b, *p)
{
  p = (i > 10) ? &a : &b

  # a = V_MAY_DEF <a>
  # b = V_MAY_DEF <b>
  *p = 3

  # VUSE <a>
  t1 = a

  # VUSE <b>
  t2 = b

  t3 = t1 + t2
  return t3
}
```

Operand Scanner

- Parses statements and expressions
  - Real operands
  - Virtual operands
- For aliased loads, pruning based on base+offset analysis of memory expression (`access_can_touch_variable`)
- Function calls receive `V_MAY_DEF` and/or `VUSE` for symbols in call-clobbered list

Virtual SSA Form

- V_MAY_DEF operand needed to maintain DEF-DEF links
- They also prevent code movement that would cross stores after loads
- When alias sets grow too big, static grouping heuristic reduces number of virtual operators in aliased references

```
foo (i, a, b, *p)
{
  p_2 = (i_1 > 10) ? &a : &b

  # a_4 = V_MAY_DEF <a_11>
  a = 9;

  # a_5 = V_MAY_DEF <a_4>
  # b_7 = V_MAY_DEF <b_6>
  *p = 3;

  # VUSE <a_5>
  t1_8 = a;

  t3_10 = t1_8 + 5;
  return t3_10;
}
```

Virtual SSA – Problems

- Big alias sets → Many virtual operators
  - Unnecessarily detailed tracking
  - Memory
  - Compile time
  - SSA name explosion
- Static alias grouping helps
  - Reverse role of alias tags and alias sets
  - Approach convoluted and too broad

Memory SSA

- Attempts to reduce the number of virtual operators in the presence of big alias sets
- Main idea
  - Stores to many locations create a single name
  - Factored name becomes reaching definition for all symbols involved in store
- Reduces
  - number of SSA names
  - number of virtual operators

Memory SSA

```
# .MEM_10 = VDEF <.MEM_0>
*p_3 = ...

# .MEM_11 = VDEF <.MEM_0>
*q_4 = ...

# b_12 = VDEF <.MEM_10>
b = ...

# .MEM_13 = VDEF <.MEM_10, b_12>
*p_3 = ...

# VUSE <.MEM_13>
t_14 = b

# VUSE <.MEM_11>
t_15 = o
```

p_3 points-to { a, b, c }

q_4 points-to { n, o, p }

At most one VDEF and one VUSE per statement

Virtual operators may refer to more than one operand

Factored stores create "sinks" that group multiple incoming names

Alias analysis in RTL

- Pure query system
- Pairwise disambiguation of memory references
  - Does store to A affect load from B?
  - Mostly type-based (same predicates used in GIMPLE's TBAA)
- Very little information passed on from GIMPLE

Alias analysis in RTL

- Some symbolic information preserved in RTL memory expressions
  - Base + offset associated to aggregate refs
  - Memory symbols
- Tracking of memory addresses by propagating values through registers
- Each pass is responsible for querying the alias system with pairs of addresses

Alias analysis in RTL – Problems

- Big impedance between GIMPLE and RTL
  - No/little information transfer
  - Producers and consumers use different models
  - GIMPLE → explicit representation in IL
  - RTL → query-based disambiguation
- Work underway to resolve this mismatch
  - Results of alias analysis exported from GIMPLE
  - Adapt explicit representation to query system