GCC Internals
Alias analysis

Google™

Diego Novillo
dnovillo@google.com

November 2007

Overview

- GIMPLE represents alias information explicitly
- Alias analysis is just another pass
  - Artificial symbols represent memory expressions (virtual operands)
  - FUD-chains computed on virtual operands → Virtual SSA
- Transformations may prove a symbol non-addressable
  - Promoted to GIMPLE register
  - Requires another aliasing pass

Memory expressions in GIMPLE

Google™

- At most <u>one</u> memory load and <u>one</u> 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

- Aliased memory referenced via pointers
- GIMPLE only allows <u>single-level</u> pointers

<u>Invalid</u>	<u>Valid</u>
`**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 (sub-variables)
- Operand scanner
  - Scans memory expressions to extract tags
  - Prunes alias sets based on expression structure

Alias Analysis

- GIMPLE only has single level pointers.
- Pointer dereferences represented by artificial symbols ⇒ ***memory tags*** (MT).
- If p points-to x ⇒ p's tag is aliased with x.

```
# MT = VDEF <MT>
*p = ...
```

- Since MT is aliased with x:

```
# x = VDEF <x>
*p = ...
```

Alias Analysis

- Symbol Memory Tags (SMT)
  - Used in type-based and flow-insensitive points-to analyses
  - Tags are associated with symbols
- Name Memory Tags (NMT)
  - Used in flow-sensitive points-to analysis
  - Tags are associated with SSA names
- Compiler tries to use name tags first

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

Alias Analysis

- Points-to alias analysis (PTAA)
  - Constraint language describes variables, operations and rules to derive points-to facts
  - Solving the system, gives sets of pointed-to symbols
  - Field and flow sensitive, context insensitive
  - 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 $*\texttt{ptr}_{42}$
  - If $\texttt{ptr}_{42}$ has NMT, use it
  - If not, fall back to SMT associated with `ptr`

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

```
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 = VDEF <a>
  # b = VDEF <b>
  *p = 3

  # VUSE <a>
  t1 = a

  # VUSE <b>
  t2 = b

  t3 = t1 + t2
  return t3
}
```

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
  - Alias sets are reduced by partitioning
  - Partitions affect representation **not** points-to results

`NMT.1` aliases `{ a b c x z l }` → 6 VOPS per store/load

`NMT.1` aliases `{ a MPT.1 l }` → 3 VOPS per store/load

Partitioning schemes

- Dynamic
  - Every store generates a different partition
  - Stores generate a single SSA name N
  - N becomes `currdef` for all the affected symbols
  - Loads are handled as usual

- Static
  - Partitions are determined before SSA renaming
  - Associations stay fixed

Dynamic partitioning

```
# .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, x }

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

Dynamic partitioning

- Advantages
  - Stores generate exactly one SSA name
  - Loads not reached by unrelated SSA names (no false conflicts)
- Disadvantages
  - Creates overlapping live ranges (OLR)
  - SSA renaming more complex
  - PHI nodes are a problem

Dynamic partitioning

```
if (...)
    # MEM_10 = VDEF <...>
    *p_3 = ...    → STORES { a b c d }
  else
    # a_2 = VDEF <...>
    a = ...
    # MEM_11 = VDEF <...>
    *q_5 = ...    → STORES { a d }
  endif
  MEM_13 = PHI <MEM_10, {a_2, MEM_11}>        STORES { a b c d }
```

- Insert one PHI node per symbol
- Defeats the factoring
- Generates even more VOPS

- Create PHI nodes for MEM
- Creates PHI arguments with multiple reaching definitions
- Leads to splitting and fixup problems

Static partitioning

- Partitions are symbols, not SSA names
- Association is done *before* SSA renaming
- Advantages
  - SSA renaming not affected
  - No OLR for virtual SSA names
- Disadvantages
  - False conflicts due to partitioning

Static vs Dynamic partitioning

```
# MEM_3 = VDEF <...>
*p_9 = ...
# a_5 = VDEF <MEM_3>
a = ...
# b_6 = VDEF <MEM_3>
b = ...
# VUSE <a_5>
... = a
```

```
# MPT_3 = VDEF <...>
*p_9 = ...
# MPT_5 = VDEF <MPT_3>
a = ...
# MPT_6 = VDEF <MPT_5>
b = ...
# VUSE <MPT_6>
... = a
```

p_9 points to `{ a b }`

Dynamic partitioning
Stores to `a` and `b` do not conflict

Static partition `MPT { a b }`
Stores to `a` and `b` **do** conflict

Heuristic for static partitioning

- Goal: minimize false conflicts introduced by partitions
  - Partition as few symbols as possible
  - Only partition uninteresting symbols

- Partitioning algorithm
  1. Gather statistics on loads and stores (direct loads/stores, indirect load/stores, execution frequency, etc)
  2. Sort list by increasing score (try not to partition symbols with high scores)
  3. Partition until number of loads/stores below threshold

Hybrid partitioning

Google

- Work in progress
- Use static partitioning to avoid the problems with PHI nodes from dynamic partitions
- PHI arguments with multiple reaching defs

```
if (...)
   # x_4 = VDEF < ... >
   x = ...
   # y_5 = VDEF <...>
   y = ...
else
   # x_7 = VDEF <...>
   x = ...
endif
MPT_10 = PHI <{x_4, y_5}, x_7>
```

New “sink” operator needed to create a new name