CS715 Project Report

Using points-to information of LFCPA in gcc

Archana & Nikhil

Note:
The points we are not sure about are marked in red.

Project Goal:

1. To inject flow and context sensitive points-to information computed by LFCPA into gcc.
2. Can gcc be benefited from these flow and context sensitive points-to information? This can be answered by looking at 'USES' of pointers.

Design Overview:
We have added functions to the LFCPA code which:

1. Display the points-to information present in gcc along with the one computed by LFCPA.
2. Inserts the flow and context sensitive information computed by LFCPA into the data structure 'pt_solution' of gcc which is a substructure of 'SSA_NAME_PTR_INFO'.

We have added print statements in other passes to look at 'uses' of pointers.

The driver function of our implementation is **'inject_pointsto_info ()'**. This function is called from driver function of LFCPA after points-to information is computed and before deletion the data structures used for storing this information.

Flow of driver function (inject_pointsto_info):

```
for every cgraph_node (function)
        for every block
            for every gimple stmt
                If (stmt is relevant) // Only for injecting points to info.
                    dump SSA_NAME_PTR_INFO
                    dump LFCPA points-to info
                    insert LFCPA pointo-to info into SSA_NAME_PTR_INFO
```

By inserting LFCPA information into gcc we mean:
For relevant statements,

get points-to bitmap of *RHS* from LFCPA data structures and copy it to points-to bitmap of LHS in gcc (SSA_NAME_PTR_INFO -> pt_solution -> bitmap vars).

COMPLETENESS of insertion process:

The insertion process which extracts points-to info from LFCPA and inserts it into SSA_NAME_PTR_INFO should not miss out any information which might be required in some optimization.

To guarantee such completeness, we should look at each and every possible use of pointers and for each such use; we should make sure that the required information has been transferred to SSA_NAME_PTR_INFO.

To achieve this goal, for each possible use of pointers, we should find out the program points i.e. gimple statements where the transfer of information should take place. Such insertion points can be definitions of that pointer (possibly, more than one). These insertion points (gimple statements) are relevant statements.

So, our goal can be stated as - *for every gimple statement defining pointer, (assuming it is used somewhere at a later program point), the points-to info of that pointer should be transferred to SSA_NAME_PTR_INFO*.

BUT, here we cannot insert any information for pointers which are used, but not defined. Such pointer uses cannot create any issue for above stated goal, as points-to information for such pointers cannot be computed. We should make sure that query for points-to info of such pointers 'points-to anything' is returned.

Note: The alias oracle (tree-ssa-alias.h) takes care of such a case and returns the appropriate information when points-to info is not available in SSA_NAME_PTR_INFO.

By looking at the existing way of storing points-to info in gcc (in SSA_NAME_PTR_INFO), we should answer few more questions for each use of a pointer. SSA_NAME_PTR_INFO allows storing of points-to information *only for SSA_NAMEs.*

Questions we should answer for each possible use of pointer:

1. Is every use always an SSA_NAME?If there is any non-SSA_NAME use, is points-to info for such a pointer available (through it's definitions)?
2. Is every use, which is an SSA_NAME always preceded by a SSA definition of that pointer? If not, we can conclude that the ssa property (every use is preceded by exactly one definition) does not always hold for pointers.
3. Does any optimization pass ask for points-to info of non-SSA_NAME pointers?

If answer to the sub-question of 1 is yes (possibility of non-SSA_NAME use with points-to info available), then we are in a situation where points-to info which might be required for optimizations is available with LFCPA, but there is no way to communicate it to other passes.

This will require us to either extend SSA_NAME_PTR_INFO to non SSA_NAME pointers or adding some mechanism for adding information of such pointers. In such a case, as per Uday sir's discussion over gcc mailing list, it can be beneficial to keep info of SSA_NAMEs and non SSA_NAMEs separate.

Possible uses of a pointer:
Pointer can be used through

1. &p (no points-to info is required)
2. p (points-to info of p is required)
3. *p, where p points-to scalar variables (points-to info of p is required)
4. *p, where p points-to non-scalar variables (points-to info of p and its pointees is required)

Any use of a pointer (in copy statements, in printf, etc.) is always one of the above 4 uses. Out of the above 4 uses, 1st use does not demand any points-to info of p and therefore, can be straightaway neglected. 3rd and 4th uses can be thought of as a cases of *p, where p may point-to any variable.

So, we will consider following two uses for discussion:

1. p (points-to info of p is required)
2. *p (points-to info of p and its pointees is required)

Now as per goal discussed above, we should make sure to transfer points-to info to gcc at definitions of these pointers.

Use 1. p:

Here we need to make sure that the points-to info of p is inserted in gcc by using definitions of p.

Use 2. *p:

Here we need to make sure that the points-to info of p and its pointees is inserted in gcc. So we should consider definitions of p as well as its pointees. Such pointers (p in use 1 and p & its pointees in use 2) can be defined in one of following ways:

p = &v
p = p2

p = *p2
*p = p2
*p = &v
p = PHI (..)

Definition of relevant statement:
There are two ways a statement can be relevant:

1. **Relevant for injecting points to information:** Such relevant statements are essentially points where we need to inject LFCPA information into gcc. The statements having (1) pointer (2) *artificial* (compiler generated) variable as their LHS are considered to be relevant statements.

2. **Relevant for subsequent optimizations i.e. statements that use inserted information:** Such statements are statements which access pointer information.

Does the definition of relevant statement completely cover all statements which are necessary to transfer all the information from LFCPA to gcc?

Points-to information can be generated by following types of statements:
Analogy used:
Local variable: local variable declared by programmer
Global variable: global variable declared by programmer
Artificial / temporary: temporary variable created by compiler (version of a variable)

Variable versions are always SSA_NAMEs
Reference: 13.3 Static Single Assignment, GCC internals documentation (4.8.0)

1. p = &v
Here p is a global or an artificial pointer and v can be a scalar variable or a pointer. If p is local pointer, a version of p (artificial pointer) will be created to which &v will be assigned.

Non-relevant:
LFCPA computes {p -> v}
But whenever this definition of p is followed by it's use (through a temporary, e.g. p_1 = p), we consider this statement (use) as relevant and process it ({p_1 -> v} is inserted into pt_solution of p_1).

2.1. *p = &v

Here p can be a global or artificial (p_1) pointer and v can be a scalar variable or a pointer. If p is local pointer, a version of p (artificial pointer) will be created to which &v will be assigned.

Non-relevant:
Let's assume {p -> x, p -> y}
LFCPA computes {x -> v, y -> v}
But whenever this statement is followed by use of x or y, we consider this use statement as relevant and process it ({x -> v} or {y -> v} is inserted into pt_solution of x or y).

2.2. *p1 = p2
Here p can be a global or artificial (p_1) pointer and v can be a scalar variable or a pointer. If p is local pointer, a version of p (artificial pointer) will be created to which p2 will be assigned.

Non-relevant:
Let's assume {p2 -> v}
Same as above

3. p1 = p2
Here p1 can be a global or artificial (p1_1) pointer and p2 is another pointer. If p is local pointer, a version of p will be created (case 2).

Case 1: p1 is a global pointer
p1 is not an SSA_NAME

Non-relevant:
Let's assume {p2 -> set s}
LFCPA computes {p1 -> set s}
But as p1 is not an SSA_NAME and points-to info of only SSA_NAMEs is of interest, we need not store this information in pt_solution. Also when such a statement is followed by use of p1 (e.g. deref p1), version of p1 get created (case 2).

Case 2: p1 is an artificial pointer (p1_1 = p2)
p1 is an SSA_NAME

Relevant:
Let's assume {p2 -> set s}
LFCPA computes {p1_1 -> set s}
We should insert {p1_1 -> set s} into pt_solutioin of p1_1.

4. p1 = *p2
Here p1 can be a global or artificial (p1_1) pointer and p2 is a another pointer. If p is local pointer, a version of p will be created (case 2).

Case 1: p1 is a global pointer
p1 is not an SSA_NAME

Non-relevant:
Let's assume {p2 -> set s1 -> set s}
LFCPA computes {p1 -> set s}
But as p1 is not an SSA_NAME and points-to info of only SSA_NAMEs is of interest, we need not store this information in pt_solution.

Case 2: p1 is an artificial pointer (p1_1 = *p2)
p1 is an SSA_NAME

Relevant:
Let's assume {p2 -> set s1 -> set s}
LFCPA computes {p1 -> set s}
We should insert {p1 -> set s} into pt_solution of p1.

Issues encountered during the project:

1. LFCPA stores points-to information with the SSA_NAME_VAR of variable instead of that variable itself.
e.g For var - p.0_1, it stores info in p.0 = SSA_NAME_VAR (p.0_1)

This doesn't create much difficulty for implementation, but can this approach of LFCPA create conflicts in two versions of the same variable?

Answer we found: It won't create any issue.

If we could create a program where a basic block contains two versions of a variable (say, p.0_1 and p.0_2) and LFCPA stores points-to info for both of them in p.0 (SSA_NAME_VAR). i.e. we want a situation where points-to information of a pointer changes without any control flow statements.
We could create such a program - test_ssa_name_var.c
Here, one basic block has two versions (p1_9 and p1_11) with different points-to info.

But fortunately, LFCPA divides the basic block further on below criteria for it’s own convenience and restores at end (call statements,on encountering dereferencing pointers, etc.). So, it would never be in a situation where two versions of a variable appear on lhs in the same basic block of LFCPA.

2. gcc computes points-to info for all pointers, but stores it (in pt_solutioin) only for pointers which are SSA_NAMEs

Reference:
a. Reply from Richard Biener (http://gcc.gnu.org/ml/gcc/2013-03/msg00254.html)
b. GCC internals documentation -
Section 13.4 Alias analysis: 2. Points-to and escape analysis:
we are only interested in SSA name pointers and
This points-to solution for a given *SSA name pointer* is stored in the pt_solution sub-structure of the SSA_NAME_PTR_INFO record.

Here we should make sure that the LHS of *all relevant statements (p1 = p2 and p1 = *p2)* must be SSA_NAMEs.

If such is not a case, some other mechanism (other than SSA_NAME_PTR_INFO) is required for adding points-to info of non SSA_NAME pointers to other passes as discussed above.

Other important points:
GCC stores a few flags in pt_solution along with bitmap of pointees. Such flags are not marked by LFCPA while computing points-to info.

Flags handled:
null
anything: It is set when a pointer points only to undef. It may be possible that a pointer points to {undef, v} because of info from different paths; but in such a case the flag is not set.
vars_contains_global

Remaining flags:
nonlocal, escaped, ipa_escaped
See example test_escaped.c.
We think that the decision of setting these flags cannot be made by looking at points-to info and should be marked while computation of info.

ipa_pta flag:
If LFCPA replaces the pass ipa-pta, it should set the flag (fun -> gimple_df -> ipa_pta) as an indication of computation of points-to info from function fun. It might be one of many such steps it should do.

Analysis:

0. Who computes the existing points-to information in gcc?
1. Examples where LFCPA points-to information is better that the existing info in gcc.
2. What kind of programs can be benefited from such a improved points-to information.
    2.1 Finding the passes which make use of the points-to info in gcc in doing optimizations
    2.2 By studying what above passes do, find programs where LFCPA points-to info leads to better optimizations.

0. Who computes the existing points-to information in gcc?

It is points-to analysis, both the IPA variant (pta pass) and the local variant (ealias and alias passes).

Local variant: Gimple passes "pass_build_ealias" (ealias) and "pass_build_alias" (alias) flags "TODO_rebuild_alias" at end and then function "execute_function_todo" from passes.c calls "compute_may_aliases ()" which eventually computes points-to information.

ipa-pta: Theoretically ipa-pta should be able to generate more precise points-to information. But because of limitations and problems in implementation, it may not be able to generate more precise information and thus, may not result in any performance improvements.

Reference:
Reply from Richard Biener (http://gcc.gnu.org/ml/gcc/2013-05/msg00010.html)

Experiments we performed:
Commented the call to "compute_may_aliases ()" from "execute_function_todo" of passes.c and replaced ipa-pta pass with pass that does nothing

Result: No points-to info is generated for any pointer (all pointers points to anything).

Result verifies that *only* IPA variant (pta pass) and the local variant (ealias and alias passes) are involved in points-to info computation.

1. Examples where LFCPA points-to information is better that the existing info in gcc:

Difference in points-to info: because of

- liveness: liveness_*.c, flow_sensitive_1.c
- flow-sensitivity: flow_sensitive_*.c
  Here difference in bitmap is result of flow-sensitivity and not liveness (except flow_sensitive_1.c).
- context-sensitivity: context_sensitive_1.c
  Here difference in bitmap is result of context-sensitivity and not liveness.

2.1 Finding the passes which can possibly make use of the points-to info in gcc in doing optimizations:

All passes that query the alias oracle (tree-ssa-alias.h) which is almost all passes doing optimization of memory accesses.

Reference:
Reply from Richard Biener: (http://gcc.gnu.org/ml/gcc/2013-04/msg00307.html)

Once we have list of such passes, we need to look at the uses of pointers for which these passes consult SSA_NAME_PTR_INFO.
Also we need to look if these passes ask for points-to info non-SSA_NAME pointers.

Passes we could find by tracing the callers of *accessor functions of pt_solution:*

1. pass_fold_builtins "fab" (fold all builtin functions)
2. pass_local_pure_const "local-pure-const" (Simple local pass for pure const discovery reusing the analysis from ipa_pure_const)
3. pass_lower_tm "tmlower" (flattening GIMPLE_TRANSACTION constructs)
4. pass_return_slot "retslot"
5. pass_dse "dse" (dead store elimination)
6. pass_tail_recursion "tailr" (Optimizes tail calls in the function, turning the tail recursion into iteration) and pass_tail_calls "tailc"
7. pass_dce "dce" (eliminate dead code) pass_dce_loop "dceloop" and pass_cd_dce "cddce”

8. pass_tm_mark "tmmark" (MARK phase of TM expansion, replaces transactional memory statements with calls to builtins)
9. pass_slp_vectorize "slp" (basic block SLP phase)
10. pass_vectorize "vect" (Loop autovectorization)
11. pass_phi_only_cprop "phicprop" (eliminate degenerate PHI nodes)
12. pass_forwprop "forwprop" (forward propagation and statement combine optimizer)
13. pass_copy_prop "copyprop" (copy propagator)
14. pass_ccp “ccp” (SSA Conditional Constant Propagation)
15. pass_vrp "vrp" (Value Range Propagation)
16. pass_object_sizes "objsz" (optimize all __builtin_object_size () builtins)
17. pass_fre "fre", pass_pre "pre" (SSA-PRE pass)
18. pass_strlen "strlen"
19. rtl_opt_pass pass_variable_tracking "vartrack" (variable tracking pass)
20. rtl_opt_pass pass_postreload_cse "postreload"
21. rtl_opt_pass pass_reorder_blocks "bbro"
22. rtl_opt_pass pass_rtl_cprop "cprop"
23. rtl_opt_pass pass_sms "sms"
24. rtl_opt_pass pass_rtl_store_motion "store_motion"

Analysis of passes:

1. pass_local_pure_const "local-pure-const":

Here this pass calls ‘ptr_deref_may_alias_global_p’ which then calls ‘pt_solution_includes_global’ *for every gimple statement*. Going up in caller hierarchy won’t help in this case.

So no conclusions can be made about pointer uses which demands points-to info from alias oracle.

2. pass_dse "dse" (dead store elimination):

This pass calls ‘ref_maybe_used_by_stmt_p’ which eventually calls ‘pt_solution_includes’ and ‘pt_solutions_intersect’ to access points-to info.

Here we could see this pass querying for points-to info of LHS for statements like:

p = &v	querying for p
p1 = p2	querying for p1
*p1 = p2	querying for p1

3. pass_copy_prop "copyprop" (copy propagator):

Calls function ‘substitute_and_fold’ which eventually calls ‘pt_solution_includes’ to access points-to info.

Here we could see this pass querying for points-to info for statements of type:
p1 = *p2

4. pass_ccp “ccp” (SSA Conditional Constant Propagation):

Calls function ‘substitute_and_fold’ which eventually calls ‘pt_solution_includes’ to access points-to info.

Here we could see this pass querying for points-to info for statements of type:
p1 = &v

5. pass_fold_builtins "fab":

Calls function ‘substitute_and_fold’ which eventually calls ‘pt_solution_includes’ to access points-to info.

Here we could see this pass querying points-to info for **‘uses’** of pointers.