CS618 Project
Porting Constraint Generation to GCC 5.2.0
..
GCC Resource Center

Ahzaz, Anshuman, Komal

December 1, 2015

Contents

1 Introduction

This report documents the work done to port the constraint generation plugin from `GCC 4.7.2` to `GCC 5.2.0`. The constraint generation plugin extracts the pointer based constraints from the GIMPLE IR during LTO (Link Time Optimization) phase and dumps them in an output file. These constraints are then used for many other pointer analysis based optimizations developed in-house.

The rest of the report documents the general approach followed to port the plugin; then the list of contraints detected by the plugin are listed with test cases; then the pattern of changes observed from `GCC 4.7.2` to `GCC 5.2.0` are listed (although not exhaustive); and at last the future work discusses the work remaining.

2 Approach Followed

This section summarises the approach followed to port the plugin. We believe this might help us to port other (if any) plugins in the future. Some of the methods are specific for this plugin. We tried understanding the code before making changes, but the direct source level changes eventually gave much quicker results. The basic approachs are enumerated below:

1. Compilation was the first problem. So we adopted the incremental approach where we added source code in logical units, files, functions etc. This helped us tackle small number of issues at a time. The functions were added by following the call hirarchy from the top. This helped us include only those functions that were reachable/used. This helped reduce the size of the code drastically.

2. How to adapt the source code was another problem. Here we compared the tree-ssa-structalias.c in `GCC 4.7.2` and `GCC 5.2.0`, and most of the differences were straight forward to understand. This worked because, the plugin was an adaptation of tree-ssa-structalias.c in `GCC 4.7.2`.

3 List of Constraints Generated by the Plugin

This section tries to enumerate all possible constraints. Each test-case and its constraints are listed below.

The programs have been created with some naming conventions of variables to make it easy to understant the type of constraints when reading. Table 1 outlines the names and their associated meaning.

3.1 Basic Pointer Constraints

Constraints related to pointer(s) to basic datatypes.

3.1.1 Single Pointer to Integer

Here the pointer is pointing to an integer. It could point to any other basic datatype, generating similar constraints.

Name	**Meaning**
i	Read ‘i’ as ‘integer’.
pi	Read ‘pi’ as ‘pointer-to integer’.
ppi	Read ‘ppi’ as ‘pointer-to pointer-to integer’. Therefore, *ppi would be a pointer-to interger.
piA, piB	Read ‘piA’ as ‘pointer-to integer A’. Capital letter suffixes like A, B, ... are used only to make the names distinct.
f	Read ‘f’ as ‘function’.
fA, fB	Functions with different names. Capital letter suffixes like A, B, ... are used to make the names distinct. Read ‘fA’ as ‘function A’.
fp	Read ‘fp’ as ‘function parameter’. Note that here ‘p’ is used after ‘f’ and hence is no more read as ‘points-to’.

Table 1: Testcase variables naming conventions

```c
// This program enumerates the semantically meaningful
// statements possible using a single pointer
// variable (pointer to int).
int* f (int *piFormal);
int main() {
    int i = 17;
    int *pi;

    pi = &i;
    i = *pi;
    *pi = 19;
    pi = pi + 1;
    pi = pi - 1;
    pi++;
    pi--;
    pi = 0;
    pi = 17;
    f (pi);

    return 0;
}

int* f (int *pifp) {
    return pifp;
}
```

Figure 1: test-01.c

Table 2: test-01 constraints

Tag	Content
Source	`pi = &i`
GIMPLE	`pi_1 = &i`
Constraint	`LHS: var id 8, ptr_ arith=0, offset 0(), type 0, name pi` `RHS: var id 9, ptr_ arith=0, offset 0(), type 2, name i`
Source	`i = *pi`
GIMPLE	`i.0_2 = *pi_1`
Constraint	`Var id 8, name pi, offset 0`
Source	`pi = pi + 1`
GIMPLE	`pi_3 = pi_1 + 4`
Constraint	`LHS: variable id 8, ptr_arith=0, offset 0(), type 0, name pi,` `RHS: variable id 8, ptr_arith=1, offset 32(), type 0, name pi`
Source	`pi = pi - 1`
GIMPLE	`pi_4 = pi_3 + -4`

Constraint	`LHS: variable id 8, ptr_arith=0, offset 0(), type 0, name pi,`
	`RHS: variable id 8, ptr_arith=1, offset`
	`18446744073709551584(), type 0, name pi`

Source	`pi++`
GIMPLE	`pi_5 = pi_4 + 4`
Constraint	`LHS: variable id 8, ptr_arith=0, offset 0(), type 0, name pi,` `RHS: variable id 8, ptr_arith=1, offset 32(), type 0, name pi`
Source	`pi--`
GIMPLE	`pi_6 = pi_5 + -4`

Constraint	`LHS: variable id 8, ptr_arith=0, offset 0(), type 0, name pi,`
	`RHS: variable id 8, ptr_arith=1, offset`
	`18446744073709551584(), type 0, name pi`

Source	`pi = 0`
GIMPLE	`pi_7 = 0B`

Constraint	`LHS: variable id 8, ptr_arith=0, offset 0(), type 0, name pi,`
	`RHS: variable id 2, ptr_arith=0, offset 0(), type 2, name`
	`null`

Source	`pi = 17`
GIMPLE	`pi_8 = 17B`
Constraint	`No constraint generated`
Source	`f (pi)`
GIMPLE	`f (pi_8)`

Constraint	`LHS: variable id 7, ptr_arith=0, offset 0(), type 0, name`
	`pifp,`
	`RHS: variable id 8, ptr_arith=0, offset 0(), type 0, name pi`

Source	`...`
GIMPLE	`D.2363_2 = pifp_1(D)`

Constraint	`LHS: variable id 10, ptr_arith=0, offset 0(), type 0, name`
	`D.2363,`
	`RHS: variable id 7, ptr_arith=0, offset 0(), type 0, name`
	`piFormal`

Source	`return pifa`

GIMPLE	`return D.2363_2`
Constraint	`Var id 10, name D.2363, offset 0`

4 Change Patterns

5 Future Work

1. The plugin source has to be logically understood.
2. A C constraint has to added. The C contraint, where a pointer is assigned an arbitrary integer (other than NULL) is not currently detected by the plugin.
3. C++ constraints have to be included.

Sr.No	Description	
1	Mapping old macros to new macros	#define VEC(A,B) vec<A> #define DEF_VEC_O(A) #define DEF_VEC_ALLOC_O(A,B) #define DEF_VEC_P(A) #define DEF_VEC_ALLOC_P(A,B) #define FOR_EACH_VEC_ELT1(A,B,C,D) FOR_EACH_VEC_ELT(*B,C,D) #define FOR_EACH_LOOP1(li,loop,C) FOR_EACH_LOOP(loop,C) #define VEC_length(A,B) (*(B)).length() #define VEC_free(A,B,C) (*C).release() #define VEC_pop(A,B) (*B).pop() #define VEC_index(A,B,C) ((*B)[C]) #define VEC_empty(A,B) ((*B).length() == 0) #define VEC_qsort(A,B,C) ((*B).qsort(C)) #define VEC_iterate(A,B,C,D) (*B).iterate(C, &(D)) #define VEC_safe_push(A,B,C,D) (*C).safe_push(D) #define VEC_last(A,B) &((*B)[(*B).length()-1]) #define VEC_alloc(A,B,C) new vec<A>() #define VEC_replace(T, V, I, V) (*V).[(I)] = (V)
2	Change in vector allocation	VEC(constructor_elt,gc) *v = VEC_alloc (constructor_elt, gc, 100);
		vec<constructor_elt>*v; vec_alloc (v, 100);
3	Change vector length api	VEC(tree, gc) *params; VEC_length (tree, params);
		vec<tree,va_gc >*params; params->length ();

4	Iterating through each basic block of each defined function.	4.7
		struct cgraph_node * cnode = NULL;
		for (cnode = cgraph_nodes; cnode;
		cnode = cnode->next)
		{// skip nodes without a body, and
		clone nodes,
		if (!gimple_has_body_p (cnode->decl) ——
		cnode->clone_of)
		continue;
		
		push_cfun(
		DECL_STRUCT_FUNCTION
		(cnode->decl));
		
		basic_block current_block;
		FOR_EACH_BB (current_block)
		print_parsed_data (current_block);
		}

		5.2
		struct cgraph_node * cnode = NULL;
		FOR_EACH_DEFINED_FUNCTION (cnode)
		{
		struct function *func;
		if(!gimple_has_body_p (cnode->decl) ——
		cnode->clone_of)
		continue;
		func=DECL_STRUCT_FUNCTION
		(cnode->decl);
		push_cfun(func);
		basic_block current_block;
		FOR_EACH_BB_FN (current_block,func)
		print_parsed_data (current_block);
		}

5	Change in safe_push api	4.7 vec<tree,va_gc>*vec; VEC_safe_push (tree_gc_vec, gc, tree_vector_cache, vec);
		5.2 vec_safe_push (tree_vector_cache, vec);
6	Change in getName api	4.7 struct cgraph_node *new_node; cgraph_node_name (new_node)
		5.2 struct cgraph_node *new_node; new_node->name ();