Neha Maddali

ComS417 Extra Credit Assignment

Question a:

Bug 1: Bug 70001 - [5 Regression] Infinity compilation time

This bug in GCC caused extremely long compilation times when dealing with certain types of nested arrays or elements with constexpr constructors. The issue was resolved by modifying the compiler's constexpr evaluator to optimize the handling of constructor initializers, thus improving compilation performance in affected scenarios.

The patch for Bug 70001 is titled 'gcc6-pr70001.patch' and it addresses a regression causing infinite compilation time. Here is the diff for the changes made by the patch:

```
--- gcc/cp/constexpr.c.jj  2016-03-08 21:04:43.050564671 +0100
+++ gcc/cp/constexpr.c  2016-03-10 12:52:04.016852313 +0100
@@ -2340,6 +2340,7 @@ cxx_eval_vec_init_1 (const constexpr_ctx
   vec<constructor_elt, va_gc> **p = &CONSTRUCTOR_ELTS (ctx->ctor);
   vec_alloc (*p, max + 1);
   bool pre_init = false;
+  tree pre_init_elt = NULL_TREE;
   unsigned HOST_WIDE_INT i;

   /* For the default constructor, build up a call to the default
@@ -2389,9 +2390,18 @@ cxx_eval_vec_init_1 (const constexpr_ctx
   {
     /* Initializing an element using value or default initialization
        we just pre-built above.  */
-    eltinit = (cxx_eval_constant_expression
-          (&new_ctx, init,
-           lval, non_constant_p, overflow_p));
+    if (pre_init_elt == NULL_TREE)
+      pre_init_elt
+        = cxx_eval_constant_expression (&new_ctx, init, lval,
+                    non_constant_p, overflow_p);
+    eltinit = pre_init_elt;
+    /* Don't reuse the result of cxx_eval_constant_expression
+       call if it isn't a constant initializer or if it requires
+       relocations.  */
+    if (initializer_constant_valid_p (pre_init_elt,
+                  TREE_TYPE (pre_init_elt))
+        != null_pointer_node)
+      pre_init_elt = NULL_TREE;
   }
       else
   {
--- gcc/testsuite/g++.dg/cpp0x/constexpr-70001-1.C.jj 2016-03-10 13:08:58.732932160
+0100
```

```
+++ gcc/testsuite/g++.dg/cpp0x/constexpr-70001-1.C 2016-03-10 13:05:53.000000000 +0100
@@ -0,0 +1,13 @@
+// PR c++/70001
+// { dg-do compile { target c++11 } }
+
+struct B
+{
+  int a;
+  constexpr B () : a (0) { }
+};
+
+struct A
+{
+  B b[1 << 19];
+} c;
--- gcc/testsuite/g++.dg/cpp0x/constexpr-70001-2.C.jj 2016-03-10 13:09:01.866889167
+0100
+++ gcc/testsuite/g++.dg/cpp0x/constexpr-70001-2.C 2016-03-10 13:07:27.000000000 +0100
@@ -0,0 +1,19 @@
+// PR c++/70001
+// { dg-do run { target c++11 } }
+
+struct B
+{
+  struct B *a;
+  constexpr B () : a (this) { }
+};
+
+constexpr int N = 1 << 4;
+struct A { B c[N]; } d;
+
+int
+main ()
+{
+  for (int i = 0; i < N; ++i)
+    if (d.c[i].a != &d.c[i])
+      __builtin_abort ();
+}
--- gcc/testsuite/g++.dg/cpp0x/constexpr-70001-3.C.jj 2016-03-10 13:09:04.700850290
+0100
+++ gcc/testsuite/g++.dg/cpp0x/constexpr-70001-3.C 2016-03-10 13:09:53.199184977 +0100
@@ -0,0 +1,26 @@
+// PR c++/70001
+// { dg-do compile { target c++11 } }
+
```

```
+#include <array>
+#include <complex>
+
+typedef std::complex<double> cd;
+
+const int LOG = 17;
+const int N = (1 << LOG);
+
+std::array<cd, N> a;
+std::array<cd, N> b;
+
+void
+foo (std::array<cd, N> &arr)
+{
+  std::array<std::array<cd, N>, LOG + 1> f;
+}
+
+int
+main ()
+{
+  foo (a);
+  foo (b);
+}
```

The patch modifies the constexpr.c file to fix the regression by reusing the return value from cxx_eval_constant_expression from earlier elements if it's a valid constant initializer requiring no relocations. It also adds three new test cases to the test suite:

1. g++.dg/cpp0x/constexpr-70001-1.C
2. g++.dg/cpp0x/constexpr-70001-2.C
3. g++.dg/cpp0x/constexpr-70001-3.C

Each of these test cases is aimed at verifying the fix for the bug.

Bug 2: Bug 114580 - Bogus warning on if constexpr

This bug in GCC caused incorrect warnings regarding the evaluation of std::is_constant_evaluated() within *if constexpr* constructs in non-*constexpr* functions. The issue was fixed by adjusting the warning generation logic to accurately reflect the behavior of std::is_constant_evaluated(), ensuring consistent and correct warnings during compilation.

The patch gcc14-pr114580.patch addresses Bug 114580 in GCC, which caused a bogus warning on if constexpr statements. Here's the diff for the changes made by the patch:

```
--- gcc/cp/semantics.cc.jj 2024-04-03 09:58:33.407772541 +0200
+++ gcc/cp/semantics.cc 2024-04-04 12:11:36.203886572 +0200
@@ -1126,6 +1126,9 @@ tree
 finish_if_stmt_cond (tree orig_cond, tree if_stmt)
```

Neha Maddali

```
 {
   tree cond = maybe_convert_cond (orig_cond);
+  maybe_warn_for_constant_evaluated (cond,
+                /*constexpr_if=*/
+                IF_STMT_CONSTEXPR_P (if_stmt));
   if (IF_STMT_CONSTEXPR_P (if_stmt)
       && !type_dependent_expression_p (cond)
       && require_constant_expression (cond)
@@ -1134,16 +1137,11 @@ finish_if_stmt_cond (tree orig_cond, tre
    converted to bool.  */
       && TYPE_MAIN_VARIANT (TREE_TYPE (cond)) == boolean_type_node)
     {
-      maybe_warn_for_constant_evaluated (cond, /*constexpr_if=*/true);
       cond = instantiate_non_dependent_expr (cond);
       cond = cxx_constant_value (cond);
     }
-  else
-    {
-      maybe_warn_for_constant_evaluated (cond, /*constexpr_if=*/false);
-      if (processing_template_decl)
-  cond = orig_cond;
-    }
+  else if (processing_template_decl)
+    cond = orig_cond;
   finish_cond (&IF_COND (if_stmt), cond);
   add_stmt (if_stmt);
   THEN_CLAUSE (if_stmt) = push_stmt_list ();
--- gcc/testsuite/g++.dg/cpp2a/is-constant-evaluated15.C.jj 2024-04-04
12:23:36.706962932 +0200
+++ gcc/testsuite/g++.dg/cpp2a/is-constant-evaluated15.C 2024-04-04 12:22:29.915882859
+0200
@@ -0,0 +1,28 @@
+// PR c++/114580
+// { dg-do compile { target c++17 } }
+// { dg-options "-Wtautological-compare" }
+
+namespace std {
+  constexpr inline bool
+  is_constant_evaluated () noexcept
+  {
+#if __cpp_if_consteval >= 202106L
+    if consteval { return true; } else { return false; }
+#else
+    return __builtin_is_constant_evaluated ();
+#endif
```

```
+  }
+}
+
+template <typename T>
+void foo ()
+{
+  if constexpr ((T) std::is_constant_evaluated ())   // { dg-warning
"'std::is_constant_evaluated' always evaluates to true in 'if constexpr'" }
+    ;                     // { dg-bogus "'std::is_constant_evaluated' always evaluates
to false in a non-'constexpr' function" }
+}
+
+void
+bar ()
+{
+  foo <bool> ();
+}
```

The patch adds a new test case g++.dg/cpp2a/is-constant-evaluated15.C which verifies the behavior of std::is_constant_evaluated() within an *if constexpr* statement, checking that the warning messages are correctly issued.

Question b:

1. Unit Testing
    a. Unit tests designed specifically for the affected functions or code paths could have helped catch the issues early. For Bug 70001, unit tests could have covered scenarios involving constant initializers and evaluated their behavior, including cases where reuse of return values occurs. By executing these tests, developers could have identified discrepancies between expected and actual behavior, indicating potential bugs.
    b. For Bug 114580, unit tests focused on *if constexpr* statements and their associated warning mechanisms could have been written. These tests could verify the behavior of std::is_constant_evaluated() in different contexts, including non-constexpr functions and *if constexpr* conditions. By evaluating the test results, developers could have detected inconsistencies or unexpected behavior.
2. Dynamic Analysis
    a. For Bug 70001, dynamic analysis techniques like runtime debugging or tracing could have used. By instrumenting the code with debug statements or using a debugger, we could have traced the execution flow through the constexpr.c file and seen the behavior of variables involved in constant initialization. Through step-by-step execution and variable inspection, developers might have noticed discrepancies or unexpected values.
    b. In Bug 114580, dynamic analysis techniques could have been useful in understanding the runtime behavior of std::is_constant_evaluated() in different contexts. With adding debug statements or using a debugger to observe the value of std::is_constant_evaluated() during execution of code containing *if constexpr* statements, we could have verified whether the function behaved as expected.