NOTE: this is a live document. Please, feel free to add comments, information and corrections. Your help is appreciated. Alternatively, you can mail me to the address below. Thanks!

Better Uninitialized Warnings

Google Summer of Code 2007 Project

Student: Manuel López-Ibáñez (Email: manu gcc gnu org)

Mentor: Diego Novillo

Contents

Synopsis

The GNU Compiler Collection warns about the use of uninitialized variables with the option `-Wuninitialized`. However, the current implementation has some perceived shortcomings. On one hand, some users would like more verbose and consistent warnings. On the other hand, some users would like to get as few warnings as possible. The goal of this project is to implement both possibilities while at the same time improving the current capabilities.

Rationale

GCC has the ability to warn the user about using the value of a uninitialized variable. Such value is undefined and it is never useful. It is not even useful as a random value, since it rarely is a random value. Unfortunately, detecting when the use of an uninitialized variable is equivalent, in the general case, to solving the halting problem. GCC tries to detect some instances by using the information gathered by optimisers and warns about them when the option `-Wuninitialized` is given in the command line. There are a number of perceived shortcomings in current implementation. First, it only works when optimisation is enabled through `-O1`, `-O2` or `-O3`. Second, the set of false positives or negatives varies according to the optimisations enabled. This also causes high variability of the warnings reported when optimisations are added or modified between releases.

What an user understands as a false positive may be different for the particular user. Some users are interested in cases that are hidden because of actions of the optimizers combined with the current environment. However, many users aren't, since that case is hidden because it cannot arise in the compiled code. The canonical example is (MM05):

```
int x;
if (f ())
  x = 3;
return x;
```

where '`f`' always return non-zero for the current environment, and thus, it may be optimised away. Here, a group of users would like to get an uninitialized warning since '`f`' may return zero when compiled elsewhere. Yet, other group of users would consider spurious a warning about a situation that cannot arise in the executable being compiled.

Other conflict is the desire by some users to emit the same warnings at `-O0` as at higher optimisation levels [JB04], while other users prefer to get as much precision as possible by discarding false positives at higher levels [RD04]. In addition, a perceived limitation of the current `Wuninitialized` is the fact that it only works with optimisation. There is no consensus on how to solve this. An approach may be to perform some dataflow analysis even without optimisation [DJ01]. However, that would hurt performance of the compiler when invoked with optimisation disabled. Other approach could warn for any potential case, even when dataflow analysis or other optimisations will easily show that it is a false positive. This latter approach coincides with request of warning about any potential usage of an uninitialized variable, even if that case cannot arise under the current compilation environment.

Proposal

From the analysis above, we can divide users into two groups with opposite requests. One group of users would like to obtain consistent, verbose warnings. The other group is interested only in cases that can actually arise in the executable being compiled, and thus, would prefer as few false positives as possible.

The proposal of this project is to divide `-Wuninitialized` into two different flags:

`-Wuninitialized=verbose`
"*Is there a code path through this function, when considered in isolation, and without being too clever, under which an uninitialized value is used?*" MM05
Produce consistent warnings across architectures and optimization levels, (and ideally releases). Warn about any potential case, even for unreachable code.

`-Wuninitialized=precise`
"*Is there a code path through this function, when compiled on this architecture with these flags, etc., for which we might actually use an uninitialized value?*" MM05
Produce the most precise warnings possible. Ideally, when more optimisations are used, more false positives are detected and not warned. This option can be used with `-O0` but it will produce many false positives. However, it will try to avoid any false positive that could be detected at that level (some cheap optimisations may be enabled at `-O0` or some limited form of dataflow analysis may be performed). Therefore, `-Wuninitialized=precise` at `-O0` is different from `-Wunintialized=verbose`, since the latter aims to be consistent while `-O0` may vary across releases or architectures.

For example, `-Wuninitialized=verbose` will warn for:

```
int i;
int j=5;
if (0)
  j = i; /* 'i' may be used uninitialized */
return j;
```

Our ability to detect some cases depends on the level of optimisation, so if we want to be consistent, `-Wuninitialized=verbose` must warn about the following always:

```
    int x, f, y;
    f = foo ();
    if (f)
      x = 1;
    y = g ();
    if (f)
      y = x; /* 'x' may be used uninitialized */
    return y;
```

In addition to this, and as a side-effect, the whole implementation of `-Wuninitialized` would be reviewed with the goal of closing as many bugs as possible [PR24639] and implementing some enhancements, like detecting access to uninitialized arrays [PR10138][PR27120]

Current Situation

Most of the code is in tree-ssa.c but the passes are scheduled in passes.c. `Wuninitialized` currently works in two phases.

- First, `execute_early_warn_uninitialized` scans BBs for SSA_NAMES that have empty definitions and emits "is used" warnings.
- The second phase, `execute_late_warn_uninitialized` repeats the first phase after optimizations and executes a second phase that looks for inputs to PHI that are SSA_NAMEs that have empty definitions. Redoing the first phase may convert some "may be used" to "is used".

Problem 1: CCP assumes a value for uninitialized variables

This probably the number 1 cause of missed warnings. CCP (Conditional Constant Propagation) assumes any value for an uninitialized variable, effectively removing uninitialized uses before the second phase can detect them. Slightly modifying the example above:

Toggle line numbers

```
int foo (int i)
{
    int j;

    if (1 == i)
        return j; /* FAIL: warning: 'j' may be used uninitialized in this function */

    return 0;
}
```

This testcase should produce a "may be used" warning in the following way. The first phase does nothing since the BB is conditionally executed, then optimizations cannot determine whether the conditional is true or false, and finally, the second phase emits a "may be used" warning. However, CCP assumes `'j == 0'` so later DCE does not consider `"return j"` to be a useful statement anymore and removes it. Thus, the second phase does not see `'j'` anymore and misses the warning.

Three alternative solutions to fix this:

(1) Warn whenever an uninitialized variable is found

Too many false warnings, since there is no way to tell whether the variable is actually used.

(2) Propagate a "uninitialized" bit and warn when folding a statement with a constant that has this bit

Here, we will only warn when a constant substitutes a variable and this constant has been merged before with a uninitialized value (see complete analysis by Diego Novillo). This should produce less false warnings. However, in the presence of loops, we cannot currently tell whether the first iteration is always executed or not.

Toggle line numbers

```
int f5(void)
{
  int x, i
  for (i = 0; i < 10; ++i)
    x = 1;
  return x;              /* { dg-bogus "uninitialized" } */
}
```

Here we get a false positive. From the SSA form is clear that GCC does not see that BLOCK 3 is executed at least once:

```
f5 ()
{
  intD.0 iD.1177;
  intD.0 xD.1176;

  # BLOCK 2
  # PRED: ENTRY (fallthru)
  iD.1177_3 = 0;
  goto <bb 4>;
  # SUCC: 4 (fallthru)

  # BLOCK 3
  # PRED: 4 (true)
  xD.1176_5 = 1;
  iD.1177_6 = iD.1177_2 + 1;
  # SUCC: 4 (fallthru)

  # BLOCK 4
  # PRED: 2 (fallthru) 3 (fallthru)
  # iD.1177_2 = PHI <iD.1177_3(2), iD.1177_6(3)>
  # xD.1176_1 = PHI <xD.1176_4(D)(2), xD.1176_5(3)>
  if (iD.1177_2 <= 9)
    goto <bb 3>;
  else
    goto <bb 5>;
  # SUCC: 3 (true) 5 (false)

  # BLOCK 5
  # PRED: 4 (false)
  xD.1176_7 = xD.1176_1;
  return xD.1176_7;
  # SUCC: EXIT

}
```

(3) Propagate a "poisoned" constant that prevents folding away the uninitialized value

Another solution would be to avoid folding `UNDEFINED` for uninitialized variables but use a special poisoned constant value. This constant value would prevent folding the uninitialized use away when it is indeed used. Later, when the poisoned constant value is found, we could warn for it. However, this will likely hurt performance, since it will prevent less constants to be propagated. For example, in principle this may hurt performance in the following case (`gcc.dg/m-un-1.c`) :

Toggle line numbers

```
sub()
{
  int i = 0;
  int j = 0;
  int k;

  while ((i | j) == 0)
    {
      k = 10;
      i = sub ();
    }

  return k;
}
```

However, in this particular case, if we initialize `k = 1`, the compiler is able to optimize the code equally good, so I suspect that it will do the same if `k` is initialized to the poisoned constant.

Problem 2: Representation issues (either IR or SSA issues)

When translating a program to the intermediate representation (IR) or to SSA, spurious uses of uninitialized variables may be introduced. For an example in the Fortran front end see PR29458.

Another problem is that the second phase may get confused by variables that have been moved / created by optimizations (FIXME:need example). Moreover, the second phase depends a lot on the SSA representation (which changes in every GCC release and with different optimization options). An additional issue is when PHI nodes do not carry the correct information about the original variables, thus giving the wrong variable name or causing a false negative (FIXME:need example).

Another issue is the representation of loops (PR43361, PR58823, PR58236 and many more). The following loops:

```
for (init; test; next) { for-body; }
init; while(test) { for-body; next; }
```

are represented by GCC as:

```
 init;
 goto eval
body:
  for-body;
  next;
  goto eval;
eval:
 (test) ? goto body : goto finish;
finish:
```

and variables within `for-body`, `next` and

are actually PHI-nodes with at least two possible values. For example,

```
# test_1 = PHI <test_2(D)(init), test_4(body)>
```

Without further analysis, GCC does not know at (-O0) that the `init` edge is always executed, so it doesn't warn. This analysis is considered to be too expensive for `-O0`, so warning in these cases requires higher level of optimization. On the other hand, Clang does warn (how?).

Problem 3: Memory references and pointers

Another important problem is that the current implementation cannot handle virtual SSA, so memory references and pointers just confuse the whole mechanism, producing both false positives and false negatives. This was partially fixed in GCC 4.4. (see PR179). However, there are still issues with PHI operands (see PR19430)

Running the alias pass before the `early_warn_uninitialized` pass seems the only way to solve the issue of memory references confusing the whole thing. (That won't solve the issue per-se but without alias info, we cannot even start to detect anything).

Problem 4: Uninitialized warnings without optimisation

GCC 4.4 enables SSA representation without optimization and, hence `Wuninitialized` can be used with `-O0`. However, the precision of the "may be" warnings without optimisation is (obviously) worse than with optimisation. In particular CCP, DCE and alias information would help to discard false positives. Maybe a limited (and very fast) form of these passes could be run without optimisation. LLVM uses this approach to warn from the front-end: https://gcc.gnu.org/ml/gcc/2008-03/msg00600.html

Problem 5: (Lack of) Predicate Analysis (a.k.a. Gated SSA / Conditional PHIs)

Testcases that will only reliably be solved by predicate analysis (Gated SSA / Conditional PHIs). (FIXME: add more explanations about)

This is the well-known problem:

```
if (p) set q;
f();
if (p) use q;
```

Most of the time, the compiler does not know at the moment of using `q` that `q` is set if `p` is true. However, many times optimisations simply get the right answer **by sheer luck** (either removing the uninitialized use or assuming that the variable was initialized): PR5035, PR20644, PR20968, PR32759, gcc.dg/uninit-5.c. Probably also gcc.dg/uninit-9.c. For example, some of these were solved by moving DCE after the latest wuninitialized pass. Moving the pass earlier will make the warnings appear again.

In some cases we are not so lucky: PR36550, PR20968, (FIXME: add more testcases).

This can only be partially solved because there may always be predicates complex enough to be beyond GCC's analysis power.

Problem 6: The Halting Problem

Not matter what we do, we will never be able to get the correct answer for every possible program. Otherwise, we will solve the halting problem. More aggressive optimisations may help to simplify code and avoid some wrong answers. One way to improve the situation would be to use predicate analysis (Gated SSA).

Patches

The following patches are in a very early stage of development:

new-wuninitialized.diff: divide `Wuninitialized` in two flags. This patch is derived from J. Law's PR24639attached

Testcases

- TODO: PR22297
- Taking the address of a variable confuses the uninitialized passes. (see PR19430)

```
/* https://gcc.gnu.org/svn/gcc/trunk/gcc/testsuite/gcc.dg/uninit-
pr19430.c */
/* { dg-do compile } */
/* { dg-options "-O2 -Wuninitialized" } */
extern int bar (int);
extern void baz (int *);

int
foo (int i)
{
  int j;

  if (bar (i)) {
    // These should do the same with respect to `j':
    baz (&j);
    // j = 1;
  } else {
  }

  return j;
}
```

- Copy of a structure partly initialized PR22197.

```
struct testme {
    int testval;
    int unusedval;
};
extern void forget (struct testme forgotten);
int main () {
    struct testme testarray[1];
    struct testme testvar;
    testvar.testval = 0;
    testarray[0] = testvar;
    forget (testarray[0]);
    return 0;
}
```

- Simple positive "may be" testcase (CCP).

```
  int x;
  if (f())
    x = 3;
  return x;
```

With `-Wuninitialized=precise` we should not warn if we can proof that `f()` returns true always, otherwise we should warn. We should always warn with `-Wuninitialized=verbose` since proving that may depend, for example, on whether `f()` is inlined.

- Testcase attached to PR18501.

```
unsigned bmp_iter_set ();
int something (void);

void bitmap_print_value_set (void)
{
  unsigned first;

  for (; bmp_iter_set (); )
    {
      if (!first)
```

```
          something ();
        first = 0;
      }
  }
```

CCP assumes that uninitialized variables can take any value and thus propagates constants and removes code. On the other hand, warning every time this happens will result in false positives (TODO: construct an "obvious" testcase for false positives that cannot be solved with Gated SSA). Diego Novillo provides a complete analysis.

- Testcase from J. Law.

```
sub()
{
  int i = 0;
  int j = 0;
  int k;

  while ((i | j) == 0)
    {
      k = 10;
      i = sub ();
    }

  return k;
}
```

This testcase is structurally equivalent to the one from PR18501. CCP cannot distinguish them, so we either warn for both or for none. A complete analysis is given by Diego Novillo.

- Uninitialized use in constructor (PR19808)

```
struct S
{
    int i, j;
    S() : i(j), j(1) {}
};

S s;
```

- Uninitialized member variable usage in constructors (PR2972)

```
struct A
```

```
{
  int f,g;
  A()
  {
    f = g;
  }
};
```

Wuninitialized and references to constants

- Uninitialized variables passed by reference as pointers to constant (PR33086). We cannot warn about this, since `use()` may cast away `const` and initialize `i`. Sorry, this is how `C` and `C++` works, not my fault.

```
void use(const int *);
void foo(void)
{
  int i;
  use(&i);
}
```

Wuninitialized and arrays

- Uninitialized array elements (PR27120).

```
int foo(int i)
{
  char buffer[10];
  return buffer[2]; /* is used uninitialized */
}
```

We currently catch this because "SRA works on the array, scalarizes the array which allows for the current initialization warning to happen" Andrew Pinski.

```
int foo(int i)
{
  char buffer[10];
  return buffer[i];
}
```

We don't catch this but we should. How?

- Uninitialized arrays passed as pointers to constant (PR10138). This is just a particular case of passing pointers to constants. We cannot warn about this because `const` can be cast away and thus the array can be initialized.

```
int atoi(const char *);
int foo()
{
    char buf[10];
    return atoi(buf);
}
```

NOTES

- What if reading an uninitialized variable is considered a side effect? (Joe Buck, 2005).

Docs and Links

GCC Internals, D. Novillo, 2007 International Symposium on Code Generation and Optimization (CGO), San Jose, California, March 2007.

JL05 Discussion about dividing and specializing `-Wuninitialized`:

PR24639 `-Wuninitialized` meta-bug.

MM05

JB04

RD04

DJ01

PR10138

PR24639attached

PR27120

GCCTestcases

An early discussion of -Wuninitialized implementation (2001)

Historical Issues

Conditional BBs

This was FIXED by the following patch https://gcc.gnu.org/ml/gcc-patches/2008-03/msg01130.html

In the first phase, the SSA_NAMES with empty definitions may happen in BBs that are conditionally executed, so a "is used" warning would be emitted were a "may be used" should be. This gets worse when the conditional BB is never executed, thus resulting in false positives. Even if GCC is able to figure whether the block is executed or not, the first phase happens way before.

An example of this issue is the testcase from PR20644

Toggle line numbers

```
int foo ()
{
    int i = 0;
    int j;

    if (1 == i)
        return j; /* warning: ‘j’ is used uninitialized in this function */

    return 0;
}
```

The corresponding SSA form (GCC 4.1.2)

```
foo ()
{
  int j;
  int i;
  int D.1280;

  # BLOCK 0
  # PRED: ENTRY (fallthru)
  i_2 = 0;
  if (i_2 == 1) goto <L0>; else goto <L1>;
  # SUCC: 1 (true) 2 (false)

  # BLOCK 1
  # PRED: 0 (true)
<L0>:;
  D.1280_6 = j_5;
  goto <bb 3> (<L2>);
  # SUCC: 3 (fallthru)

  # BLOCK 2
  # PRED: 0 (false)
<L1>:;
  D.1280_4 = 0;
  # SUCC: 3 (fallthru)

  # BLOCK 3
  # PRED: 1 (fallthru) 2 (fallthru)
  # D.1280_1 = PHI <D.1280_6(1), D.1280_4(2)>;
<L2>:;
  return D.1280_1;
  # SUCC: EXIT

}
```

In `BLOCK 1`, `j_5` is used and it has an empty definition. However, the whole block is only executed if predicate 0 is true, which it is not in this case. Nonetheless the current approach is unable to detect this. A solution would be to warn here only about blocks that are reached unconditionally (`FALLTHRU`). Then, in the second phase, distinguish between conditional and unconditional blocks ("may be used" vs. "is used"), hoping that optimizations would help to distinguish whether blocks are executed or not.