Reprinted from the

Proceedings of the GCC Developers' Summit

June 17th–19th, 2008
Ottawa, Ontario
Canada

Improving the precision of GCC's debug information

Michael Matz
SuSE Labs
matz@suse.de

Abstract

Debug information of optimized code is becoming more and more important. A primary example is the use of systemtap (or similar means) to inspect whole systems while they run in production mode (and hence with optimized code). Currently GCC loses quite some information during its optimization passes, some of that by necessity (e.g., whole subexpressions can be optimized out) and other just because of missing tracking of values. Of particular importance is the mapping from user-named variables to values. The SSA infrastructure gives us a way to name each value produced and bind them to user variables. That mapping can be maintained over optimizations into the RTL representation and from there converted to debug information. This work is being carried out on the var-mappings branch, and here we present that work, the infrastructure, and possible future enhancements.

1 Introduction

One of the most important aspects of debugging is introspection,[1] i.e., the possibility to learn the values of different entities of a program. The most useful entities are user variables—be they global, static, or automatic—as these are the ones with the most direct relationship to the source code.

Hence, one goal of generating debug information is to provide a most precise picture of what the value of a user variable is, at preferably all points during the program flow. Traditionally GCC did fairly well with this in absence of optimization, as the mapping from user-declared variables to places (in either registers or memory) doesn't change often inside a function body.

[1] Other aspects are possibility of state changes, like changing values or control flow.

With optimization, though, solving this problem is more difficult, as a user variable can be placed in different registers (and memory) at different points in time. Unfortunately debuggable-but-optimized code becomes more and more important, often not because of the wish for traditional debugging, but for introspection (e.g., whole system tracing).

In Section 2 we will look in detail at the several aspects of the problem at hand and try to provide means for how to measure them and formulate some goals. In Section 3 we propose a solution for these goals, whereas in Section 4 we outline some possible enhancements.

2 Anatomy and Taxonomy

2.1 Details of the problem

There are many transformations done to a program during optimization that affect the ability to debug it. Furthermore different people have different expectations of what a debugger should be able to do, so the first step is to define what precisely we want to (and what we don't want to) achieve.

For this purpose we look at several functions with their optimized (or transformed) counterparts exemplifying some situations that often happen (obviously these are contrived examples):

```
int foo1(int j) {
  int I = j * 2;
  return I;
}

int foo1_optimized(int j) {
  int temp = j + j;
  return temp;
}
```

foo1 is an example where the optimizer simply replaced a user variable I by a temporary; i.e., it's unreasonable for a debugger to refuse to output the value of I, as it's readily available somewhere.

```
int foo2(int j) {
  int I = j * 2, J = j + j;
  return I + J;
}

int foo2_optimized(int j) {
  int temp = j + j;
  return temp * 2;
}
```

foo2 shows an example where two user variables get replaced by the same temporary. Again a debugger should deal with this situation and give an answer if asked for either I or J.

```
int foo3(int j) {
  int I = j * 2;
  return I + j;
}

int foo3_optimized(int j) {
  int temp = j * 3;
  return temp;
}
```

In foo3, on the other hand, no temporary contains the value of user variable I. It would have to be computed, which is possible in this example, but not in general (not all functions are injective). For this work we decided to not generate debug information in this case.

```
int foo4(int j) {
  int I;
  callit(i * 2);
  I = i * 2;
  return I;
}

int foo4_optimized(int j) {
  int temp = i * 2;
  callit(temp);
  return temp;
}
```

The last example, foo4, shows a situation similar to the first two, but this time the point of definition of values for user variables is moved to different places, i.e. their points of life and death changed. We think it is reasonable for a debugger to give a value for user variable I at a program point, even when it hasn't yet been defined in the original source (or when it isn't needed anymore)—in this case, before the call to callit. That is a side effect of optimization and doesn't inhibit debugging capabilities (after all, it soon will get that very value also in the original program order). So we don't especially care about this situation.

So, in summary, if the value of a user variable happens to be available directly in some place we want to reflect this in the debug information, not necessarily taking into account whether the user variable really is defined at that program point. If the value can merely be expressed as an expression we don't want to say so.

2.2 Connection to GCC

This work doesn't exist in a void; therefore, we need to build a connection from the above phrases to concepts of GCC. In particular, we need to decide what we mean by *values* and *user variables*: A **user variable** simply is the VAR_DECL or PARM_DECL coming directly from the front-end. In the final debug information **values** are often identified with **places**, which are either *constants*, *registers*, or *memory slots* (i.e., debug information describes where a value can be found, not the value itself).

In GCC we have two large intermediate representations: GIMPLE and RTL. During GIMPLE registers are represented as temporaries (that fulfill the is_gimple_reg predicate and will be rewritten into SSA form), during RTL as pseudo registers; constants can represent themselves.

Memory becomes interesting only very late in compilation. Up until the register allocator, memory accesses in the intermediate representation correspond directly to memory accesses which also exist in the original program source. They are never generated artificially. A user variable that happens to be placed in memory by the front-end (e.g., for a global variable) never changes that place (it can be loaded into registers, though); and the place is noted in the corresponding DECL node.

The register allocator can introduce new memory accesses (to space on the stack) when it has to spill pseudo-registers to make all temporaries fit the CPU register file.

All of these stack slots can be associated with the register they contain.

Therefore it is enough if we limit ourselves to constants and registers. In fact for this paper we are concentrating on registers only (but see Section 4) and we interchangeably will use the terms register and temporary (and sometimes, SSA name) for them.

By dealing only with GIMPLE values (during GIMPLE), we automatically constrain ourselves to user variables of types that really can be placed into registers, and only those pose problems. This includes the scalar types, but also some aggregates (small structures or vectors). Variables of more complicated types will be placed into memory from the start, and are dealt with automatically as described above.

2.3 Current situation

Much of the infrastructure to generate precise debug information already exists in GCC. For instance, there is a scope tree (mapping back instruction locations to the original program scopes, that itself contains a list of user variables declared therein). The debug information generators (in particular the one for DWARF2) are capable of generating location lists for each declaration (mapping a *variable, program* point pair to a place).

Furthermore, we have the possibility to control these emitters via pseudo-instructions, announcing a change of location for variables. These pseudo-instructions are generated on the (mostly) final instruction stream by the `var_tracking` pass and can make use of any information available at that point. This includes annotations on `MEM` and `REG` expressions (giving a declaration for the described place) and solving a data flow problem on the learned initial situation.

The missing things are basically that `MEM` and `REG` expressions can be associated with only one user variable, and us not maintaining the associations between user variables and temporaries—thereby having lost much of the necessary information from the input to `var_tracking`.

3 Solution

The goal now should be clear. We want to build a mapping from user variables to registers, maintain it as much as possible throughout optimization, and finally use it while generating the debug information.

It's not enough for this mapping to simply map from a declaration to a set of registers; we wouldn't know which register to look at at a certain program point. We effectively also need to store a partitioning of the instructions. During the course of optimization that partitioning constantly changes, and we can't easily know how when we only have a map starting from declarations. The optimizers don't know (and shouldn't know) about user variables, so while doing their transformation they need to have a way to go from the primitives they work with (SSA names) to that mapping.

Hence one prerequisite for maintaining the first mapping is also a mapping from SSA names to user variables.

3.1 The idea

The key insight for easy maintenance of the decl-to-place mapping now is that we can use the name-to-decl mapping *instead*. An SSA name effectively provides a set of instructions at which it is has a value (namely all that are dominated by its definition). And SSA names also correspond to exactly one value each (they are defined only once), so a change in value will give rise to a new SSA name, a possibility to associate a different set of user variables with that new value.

Given a name-to-decl mapping, we can construct a decl-to-place mapping trivially: Given a user variable U and a program point p, search the nearest dominating SSA name definition d that is associated with U. The value of U at p is contained in d. Additional precision is reached by checking if U actually is active in a scope that contains p.

The nice thing about having the mapping in this direction is that the instruction partition is implicit in the dominator relationship of SSA name definition, and hence doesn't need to be maintained explicitely. If any optimizer moves such definition to some other place (and it was a valid transformation) it still is associated with U (that happens to now look initialized earlier or later).

So, we introduce a general map from SSA names to a set of `DECL`s.

3.2 RTL

The important property of the name-to-decl mapping is that we have unique entities that represent changes in value to which we can attach associated variable declarations. In the RTL intermediate representation, pseudo-registers represent the places; however, unlike SSA names, a pseudo-register can be assigned to multiple times and hence represent multiple values (at different program points), so we can't use them to maintain the same mapping.

Another property of the SSA names was that most of the optimizers didn't need to be aware of them being associated with user variables. So we'd like to find something similar in RTL: unique, self-contained, and representing value changes.

We decided to use the `SET` RTL construct for this. It represents exactly one side-effect, is unique (because RTL trees are unshared), and is reasonably self-contained. Another natural place would be the `INSN` itself, but there the handling is already more complicated, as RTL instructions can have multiple side-effects (and hence `SET`s), and often RTL optimizers don't deal with whole instructions (requiring representing the variable associations on the side).

Hence we introduce a general map from `SET` to a set of `DECL`s.

3.3 SSA to RTL

One difficulty occurs while going from GIMPLE SSA form to RTL form. First we go out of SSA form (losing the uniqueness of SSA names), and second, we build long expressions out of several smaller instructions for the benefit of the RTL expander. The latter replaces individual temporaries (that now might be associated with user variables) with whole expressions.

This phase is very short and doesn't contain any other transformations, so a special-case solution is sensible. What we do is to move the associations from SSA names to their right-hand side expressions. These are unique and are the input to the RTL expander that can lookup associations for every expression it expands and move them into the generated `SET`s.

This also solves the problem of nested expressions, as those can be associated with user variables, also. They will be expanded recursively and turn into a number of `SET`s, each of them possibly then associated with set of user variables corresponding to the just-expanded expression.

For example, given this short program:

```
static int bar(int j) {
  return j + 1;
}
int l;
int foo5(int i, int k) {
  l = bar(i*k);
  return l;
}
```

we want to retain the association between `j` and the expression `i*k`. After inlining and during expansion, the expander will be given a tree for the whole expression $i*k+1$; therefore, we need to have a general association between (sub)expressions and variable names.

3.4 Maintaining the mapping

Now that we know what our mapping looks like, we need to say how we maintain it during optimizations, which will be surprisingly simple.

Once an SSA name is associated with a user variable, it always will be. All optimizers that simply work with the `SSA_NAME` tree, moving it around or placing it into new expressions, don't need to do anything special—at most the program points at which a value is available change, but that never loses the value completely.

When a new SSA name is generated, it doesn't immediately need to be associated with a user variable: either it's a completely new value not associated with any variable, then it never will be, or it's a copy of a value already associated—in that case, both are equivalent and the source of that copy can be used to maintain the mapping just fine.

The only situation we need to deal with is when the last reference to a place is removed, which can only be a definition. When this happens and the definition was a copy, then we make the source SSA name now also be associated with all user vars of the to-be-deleted destination SSA name. When the transformation is correct, then also this moving is correct, and possibly only moves the point of definition closer to the function entry.

This can be seen in the following example:

```
int foo6(int j) {
  int I;
  callit(j * 2);
  I = j + j;
  return I;
}
```

this will be expanded to roughly this body:

```
temp1 = j * 2;
callit(temp1);
temp2 = j + j;  // vars(temp2) = I
```

where `temp2` is associated with `I`, while `temp1` has no associations. When common expressions are optimized away, the definition of `temp2` will transform into a copy (from `temp1`) that in turn will be removed by dead code removal:

```
temp1 = j * 2;
callit(temp1);
temp2 = temp1; // vars(temp2) = I; but dead
```

At the point of removing the definition of `temp2`, we will associate `temp1` with variable `I`, retaining that information. The noticeable effect to the user is that `I` now seems to be initialized already before the call (but to the correct value), which seems a small price to pay.

The situation is similar in RTL form. As long as the `SET`s are handled *en bloc*, nothing needs to be changed. When we remove a `SET` we again need to handle copies specially. This time we need to find the dominating `SET`s that define the right-hand side of the to-be-deleted one to move the associations to.

3.5 Implementation

We implemented the ideas from Section 3.1 *et seq.* on the `var-mappings` branch. All `DECL` nodes already have unique numbers assigned to them, so we implement the target set of our mappings simply as a `bitmap`. To facilitate lookups we also have a map `decl_for_uid_map` from these IDs to variable declarations.

The mapping itself is implemented via `uid_bitmap_map` and `tree_bitmap_map` for the tree-SSA side, and via a new slot in the RTL `SET` expression on the RTL side (referring to the uid bitmap).

As expected there were only limited changes necessary to the various GCC passes to maintain this mapping. The most prominent ones are `release_defs` where SSA names are removed, and `setup_one_parameter` to connect formal parameters of inlined functions to their SSA temporary.

Most of the transformations on RTL (that affect individual instructions) are done via `validate_change`, often changing the operands of `SET`s in place. This doesn't invalidate the variable associations. The only place that really can remove `SET`s is `remove_insn`. At the time of this writing the move of associations is only partially implemented here.

One place that generates new `SET`s from old ones is the instruction splitter, that simply can move variable associations to the last set.

Finally we also had to amend the `var-tracking` pass to make use of that mapping in the obvious way (when we encounter a `SET` with associations, we emit annotations that the destination of it now contains those variables).

A result of this work can be seen e.g. here (from the `final_cleanup` dump, i.e., before RTL expansion). The input function

```
int bar(int j) {
  int jj = j / 41;
  int kk = jj + 1;
  int ll = kk;
  return ll;
}
```

is transformed into

```
int bar(int j) {
  return j / 41 E{ jj } + 1 E{ kk ll };
}
```

The notation `E{foo}` means that the expression before it is associated with the mentioned name(s). Furthermore the notation is omitted when the expression is just the variable itself (e.g., when the variable was not decomposed into new temporaries). In this example, that would be `j`.

4 Future Work

There is still work to do. It would be useful to also handle constants (one idea would be a map from constants to variables that could be used when a variable has no other place).

It would be even more useful to have a better test suite for debug information, and some means to measure the precision of it. One could try using the gdb test suite for this.

Furthermore, there are undoubtedly more places in GCC that unintentionally lose variable associations, but could easily be fixed. Not many of them are visible when looking at examples that are simple enough to analyze by hand, so also for that a test suite would be very helpful, or at least an automatically checked metric (e.g., over how many instructions a user variable is available in the debug information).

One extension to the mapping could potentially provide some more precision without destroying the basic premise of easy maintenance: if the map targets weren't only variable declarations, but pairs of scope block and var-decl, we may be able to better confine these associations (as explained in Section 3.4, we sometimes extend location lists beyond their original definition points).

5 Conclusion

We presented a way to maintain the association between user-declared variables and compiler-generated temporaries (SSA names, pseudo registers). We heavily built on the existing infrastructure of GCC, and created a mapping that is maintained nearly automatically by many optimization passes, thereby reducing the overall amount of necessary changes.

Due to the way we built the mapping, it's also trivial to see that it doesn't affect code generation in any way, so one invariant of debug info generation (same code with or without it) is easily satisfied.