Consistent Views at Recommended Breakpoints

Alexandre Oliva
Red Hat
aoliva@redhat.com

Abstract

When users of source-level debugging and monitoring programs choose to inspect the state of the program at a certain line number, it is desirable that effects of logically prior source statements be in the picture presented to the user, and that effects of subsequent statements not be in it. However, as optimizations change, drop and reorder code fragments, concerted effort is required of the compiler and the debug information consumer to select appropriate inspection points, and to reconstruct coherent pictures for the user.

We propose a robust method for the compiler to select and annotate meaningful, consistent inspection points, and debug information extensions to support multiple coherent pictures at a single program location. They enable debug information consumers to stop at inspection points that could not be represented otherwise, and to advance to the user-visible states of subsequent source lines, even in the absence of intervening executable instructions.

1 Introduction

Much effort has been put into maintaining, throughout compilation, the association between source variables and their run-time locations or values, as well as between executable code fragments and their corresponding source-code line numbers.

However, it is often the case that, because of optimization, code fragments from one statement run before previous statements complete. Thus, when debug information consumers wish to inspect the state of the program at a certain line, the picture they expect may not be available at the selected inspection point. Indeed, deepending on instruction reordering, the expected picture may not be available at any instruction generated out of the chosen statement.

In earlier work[2], we have introduced Variable Tracking at Assignments (VTA), i.e., annotations that bind user variables to expressions that denote their location or value, and whose ordering is maintained unchanged during compilation, regardless of instruction reordering. As such, these annotations offer a progressive view of the side effects of source statements, as expected from the source-level point of view.

In this work, we take advantage of the stability of these annotations over reordering optimizations, and introduce additional annotations to provide the missing information, so as to select the ideal inspection point for each source line or statement, so as to offer a view of the program state that matches the expectations of an non-optimized compilation.

This article is structured as follows. In Section 2, we cover some of the background of debug information generation and variable tracking at assignments. Section 3 describes the newly-proposed annotation to mark the frontiers between statements. Section 4 justifies and proposes extensions to existing debug information formats[1] to enable multiple inspection points per program location, and Section 5 offers some advice to debug information consumers. Section 6 concludes with suggestions and plans to deal with situations that are not covered in the present work.

2 Background

There are two aspects of debug information we are concerned with in this article: line numbers and variable locations. They're described in independent sections in standard debug information formats such as DWARF. They're designed so that debug information consumers can find, for a given PC (program counter, the address of an executable instruction in the program), the corresponding source location and the run-time location of available user variables.

2.1 Line number information

Line number information is represented in DWARF using very compact and extensible line number programs. These programs tell debug information consumers how to build a table with one row per executable instruction, with columns indicating address and index (for machines that pack multiple instructions into a single word), source file, line and column the instruction pertains to, the ISA (for executable formats that support multiple instruction sets), whether the instruction is the beginning of a source statement, of a basic block, of a function epilogue, whether it is the end of a function prologue.

Line number programs take the form of a sequence of opcodes. Various standardized opcodes set values for specific columns in the next row of the table. Others advance the address, index and line number by fixed amounts, completing a row and advancing to the next entry in the table.

The encoding is extensible in that opcodes to set new columns can be added in a way that does not prevent debug information consumers that are not aware of it from parsing the remaining line number information. For example, DWARF 4 standardized a discriminator column, that associates an instruction to a block identifier, a number arbitrarily assigned by the debug information producer.

Although it is possible for a compiler to generate line number programs, computing instruction lengths, alignments and padding is easier done by the assembler. To this end, assemblers often offer pseudo-opcodes such as `.file`, to name source files, and `.loc`, to indicate the points of line number changes, as well as other columns for the line number table.

GCC associates source location information with tokens as they are parsed, and propagates these locations to declaration and expression trees, then to Generic stmts, GIMPLE tuples, and RTL insns, maintaining them reasonably accurate as optimizations split, combine, and reorder executable elements of the various compiler internal representations. As assembly code is generated for RTL insns, `.loc` directives are issued as mandated by the location information present in the insns.

Some loss is experienced when optimizations restructure the code, e.g., when portions of different blocks, identical in effect but not in line numbers, are unified into a single block. Line number programs in DWARF are not geared towards supporting scenarios in which the same instruction corresponds to multiple source code blocks, even more so if it would be desirable to trace which source block a specific execution of the instruction pertains to. This is a problem that we're saving for future work.

2.2 Variable location information

Debug information entries for user variables appear in the linearized tree data structure that represents a compilation unit. Each node in the tree stands for an entity such as a module, a namespace, a subroutine, a lexical scope, or a data constant or variable. It may have children nodes, as well as multiple attributes. A node that stands for a user variable may have attributes for a constant value or a location. The location attribute may be represented as a single location descriptor, applicable to the entire lifetime of the variable, or as a location list.

A location list is a sequence of tuples that specify an address range and a location expression. The range is given as a pair of (relative) addresses: the lowest address at which the expression applies, and the lowest subsequent address at which it no longer applies. Multiple tuples may cover the same address range, implying the variable is concurrently available at all the given expressions.

A DWARF version 3 location expression names a register, or a memory location, specified as a computation out of constants, registers and other memory locations. The computation is described with standardized opcodes on a stack machine.

DWARF version 4 introduced a new opcode that enables a location expressions to yield a computed value a user variable would hold if the computation hadn't been removed by optimizers. Debug information consumers that do not implement this new opcode disregard the entire expression, but they're no worse off than when such an expression couldn't be emitted.

In order to generate variable location information, GCC associates RTL expressions that represent (pseudo) registers and memory locations with corresponding variable declarations. At the end of compilation, the Variable Tracking pass performs global analysis to infer

the dynamic location of variables, as they're loaded from memory, modified or copied between registers, and stored back in memory. At the end of all optimizations, it emits notes that bind user variables to the inferred locations, and these notes are later used to generate location descriptors and lists.

As more optimizations were introduced before RTL expansion, this association became highly unreliable for variables that didn't live permanently in memory. To fill the gap, we introduced Variable Tracking at Assignments (VTA), in GCC 4.5. It also involves notes that bind user variables to expressions that denote their location (or value), emitted after each assignment (thus the name) to variables whose locations may vary over their lifetimes. Like line number information, these notes are introduced early on in compilation, before the internal representation diverges from source code, and maintained accurate throughout optimizations with little effort from optimizers. These notes feed the global analysis performed by Variable Tracking, so that location notes can be emitted not only at assignment points, but also when e.g. registers are reused for other purposes, but the value of the variable is still computable or available elsewhere.

3 Statement Frontier Annotations

One particularly important feature of VTA bind notes is that they offer a progressive view of the computation, as specified in the source code. Optimizers are not supposed to reorder these notes: even if an assignment operation is moved or optimized away, the bind note remains in place, adjusted as required to bind the user variable to the user-expected value expression.

As such, for programs whose computations amounted to nothing but assignments to user variables whose locations might vary over their lifetimes, select VTA bind notes would be excellent markers for the end of source statements. Sadly, most non-trivial programs call subroutines, and they often use variables that don't require dynamic location tracking, so relying exclusively on VTA bind notes to denote the frontier between one source statement and the next would not work.

We have thus considered emitting notes at the end of statements that didn't end with a VTA bind note. However, debuggers and their users seldom think in terms of ends of statements. Users, influenced by debuggers' interfaces and vice-versa, tend to think in terms of "stop at line N" to inspect the effects of line $N-1$. Indeed, debug information formats such as DWARF have support for marking the beginning, not the end of a statement in a line number table. Moreover, the end of a statement marker would seldom even have an instruction pertaining to that statement to bind itself to: its row in the line number table would be followed by a row number entry for the same address, denoting the source location of the subsequent instruction.

Table 1: Source file f.c

```
1:int f(int a, int b, int c, int d) {
2:  int x = a + b;
3:  int y = c / d;
4:  x -= y;
5:  return x;
6:}
```

In fact, GCC 4.5 emits (`is_stmt 1`) markers in line number information every time the line number changes. However, these markers, as they are implemented nowadays, seldom form a coherent picture with the program state exposed by variable location information. Our proposal is to issue beginning-of-statement notes early on in compilation, to keep them from being reordered with respect to each other and debug bind notes, so that they become part of the progressive view, and finally, to use them to decide when to issue `is_stmt` markers.

To illustrate the difference, consider the simple function depicted on Table 1.

Compiled for a superscalar machine that passes arguments on the stack, the scheduler might move the loads and the division up to hide their latencies, and we'd generate code such as that of Table 2. Register annotations indicating source variables are displayed in parentheses after register names.

GCC 4.5 would generate debug information as depicted on Table 3. Location information is denoted in # comments. Note how the statement marker for line 3 appears before that of line 2. Further observe how the computation from line 4 was substituted into the return value computation in line 5, so that no code remained in line 4 proper. If you were to request a debugger to stop at line 4, it would stop at line 5, at which point the initial value of `x` would no longer be available.

Table 2: Optimized f.c

```
f:
  r5(c) <-   *(sp+12)
  r6(d) <-   *(sp+16)
  r2(a) <-   *(sp+ 4)
  r3(b) <-   *(sp+ 8)
  r7(y) <- r5(c) / r6(d)
  r4(x) <- r2(a) + r3(b)
  r1(x) <- r4(x) - r7(y)
  ret
```

Table 3: Optimized f.c with GCC 4.5 debug info

```
f:
 .file 1 "f.c"
 # a => *(sp+ 4)
 # b => *(sp+ 8)
 # c => *(sp+12)
 # d => *(sp+16)
 .loc 1 3 is_stmt 1
  r5(c) <-   *(sp+12)
  r6(d) <-   *(sp+16)
 .loc 1 2 is_stmt 1
  r2(a) <-   *(sp+ 4)
  r3(b) <-   *(sp+ 8)
 .loc 1 3 is_stmt 1
  r7(y) <- r5(c) / r6(d)
 .loc 1 2 is_stmt 1
  r4(x) <- r2(a) + r3(b)
.L0:
 # x => r4(x)

.L1:
 # y => r7(y)
 .loc 1 4 is_stmt 1
.L2:
 # x => r4(x) - r7(y)
 .loc 1 5 is_stmt 1
  r1(x) <- r4(x) - r7(y)
.Lret:
  ret
.Lend:
```

Table 4: Optimized f.c with proposed debug info

```
f:
 .file 1 "f.c"
 # a => *(sp+ 4)
 # b => *(sp+ 8)
 # c => *(sp+12)
 # d => *(sp+16)
 .loc 1 3 is_stmt 0
  r5(c) <-   *(sp+12)
  r6(d) <-   *(sp+16)
 .loc 1 2 is_stmt 0
  r2(a) <-   *(sp+ 4)
  r3(b) <-   *(sp+ 8)
 .loc 1 3 is_stmt 0
  r7(y) <- r5(c) / r6(d)
 .loc 1 2 is_stmt 1
  r4(x) <- r2(a) + r3(b)
.L0:
 # x => r4(x)
 .loc 1 3 is_stmt 1
.L1:
 # y => r7(y)
 .loc 1 4 is_stmt 1
.L2:
 # x => r4(x) - r7(y)
 .loc 1 5 is_stmt 1
  r1(x) <- r4(x) - r7(y)
.Lret:
  ret
.Lend:
```

Now, consider the alternate `is_stmt` locations displayed on Table 4. The moved-up loads are no longer the recommended breakpoints. Instead, they are close to the bind annotations that denote the progressive side effects expected from the source code. When users stop at a breakpoint and find themselves at a certain line, they will be able to observe the completed effects of previous statements.

4 Debug Information Extensions

Unfortunately, it is still the case that no instruction remained between the recommended breakpoint for line 4 and that for line 5, so asking for a breakpoint at line 4 would land you at line 5. In a way, it is now worse, because the lack of instructions applies to the recommended breakpoint for line 3 as well. This is not an unusual situation. Scheduling an instruction from one line right after a marker for a different line is not unusual either, and this could land you not at a subsequent line, but also at a previous line, which could be very confusing.

These realizations led to the conclusion that we needed means to tell apart different source-level states at the same PC. Indeed, GDB could already present different pictures at the same PC, in cases of inlined functions, emulating the different frames that would have been constructed should the function not be inlined. We just had to take this notion of multiple views per PC a step further.

The goal could be stated as enabling the compiler and the debugger to behave as if a `nop` instruction was emitted after each `is_stmt` marker, as far as breakpoints and single-stepping are concerned, without disruption for debug information consumers that did not support the extensions required to this end, and while still permitting the division of work between compiler and assembler in generating debug information.

It was clear that extensions would be required. Both line numbers and location lists are keyed off of instruction addresses, so additional information would be required to establish the link between line numbers and location expressions pertaining to different views at the same PC.

So say we introduce view identifiers, or view for short. Adding a new column to the line number table to carry it is easy, but adding the information to location lists is not so trivial. For example, testing view numbers in location expressions would invalidate the expressions for debug consumers, even though, lacking support for multiple views per PC, using the ranges alone would be correct. Modifying the location list format to support view numbers as part of the ranges would render the entire location lists incompatible. Adding another attribute to variables, to point at a view number table, would work, but it would be a bit wasteful in terms of space. The solution we chose: adding the view numbers right after the end of a location list, with a new flag attribute, say `DW_AT_location_has_views`, to indicate their presence.

No disruption to existing debug information consumers, check. But how about enabling the compiler to generate location lists and the assembler to emit line numbers? Surely, if the compiler does all the work, it can choose to its liking the view numbers for line number table rows, and then reference them at the view addends at the end of location lists.

It should be noted that view numbers need not be globally unique, not even within a compilation unit. As long as different views at the same PC are assigned different numbers, it could be made to work. However, it is more convenient if view numbers can be regarded as a less-significant, fractional portion of the address, i.e., that they are ordered and can be compared according to the logical execution order. Compilers can thus reset the view number whenever the instruction address changes.

Assemblers may accept view numbers from the compiler, as additional operands to `.loc` directives, but since the assembler knows better than the compiler when addresses change, the compiler can leave it up to the assembler to select view numbers in `.loc` directives: for every such directive without any intervening address change, the view counter is incremented. Labels defined afterwards inherit the view counter in effect, an assembler pseudo-opcode or functional operator can be introduced to output the view counter associated with a label as part of location lists addends.

Given these constraints, labels `.L0`, `.L1`, `.L2`, `.Lret` and `.Lend` can be assigned views number 0, 1, 2, 0 and 0, respectively, so that variable `x` is marked with say `DW_AT_location_views`, and the location list can be emitted as depicted on Table 5.

Table 5: Location list for variable `x`

```
.LLSTx:
 .long .L0 - f, .L2 - f
  .value 1
  .byte DW_OP_reg4
 .long .L2 - f, .Lend - f
  .value 6
  .byte DW_OP_breg4, 0,
        DW_OP_breg7, 0,
        DW_OP_minus,
        DW_OP_stack_value
 .long .Lret - f, .Lend - f
  .value 1
  .byte DW_OP_reg1
 .long 0, 0
 .view .L0, .L2,
 .view .L2, .Lend
 .view .Lret, .Lend
```

5 Usage

Being able to stop a program at any line that contained source code, and inspect its state so that the effects of all prior statements and none of the effects of subsequent statements are visible, is certainly desirable for debuggers such as GDB, but it is even more important for monitors such as SystemTap. Such tools ought to use view information as soon as it is available.

Multiple views per PC enable debuggers to advance the view to another line without actually changing the underlying state of the program. The ability to step from one line to another even when no code remained between their recommended breakpoints may offer users a closer debugger experience to that of a non-optimized build, which some users may welcome. Others may be disturbed if a request to let the program run for a bit doesn't, and the PC remains the same. We suggest debuggers to offer both possibilities.

6 Future Work

This proposal brings us closer to the ability to debug and monitor optimized programs as closely as possible to the non-optimized counterparts. However, a number of issues remain to be addressed when it comes to optimizations that restructure the code, say, by factoring common code out of multiple converging blocks, as mentioned in Section 2.

Saving in debug temporaries the conditions used to decide between multiple blocks is something we already envisioned, to deal with conditional binds in blocks combined by if-conversion passes. If this proves effective, it may pave the way to applying such conditionals also to select the right view for a PC address.

On a breakpoint specified only by its PC address a debugger may not be able to associate the intended source line to it. The debugger would have to default, for example, to the first view for that PC.

Being able to inspect optimized program state in a way that better reflects the expectations that would be met by non-optimized programs should be a significant improvement to the usability of debuggers, system monitors and other tools that rely on debug information to inspect the state of the program.

Acknowledgments

The idea of being able to stop at source lines that had been completely optimized away was first suggested by Red Hat colleague Jan Kratochvil. The infrastructure required to support this turned out to be essential for statement frontier annotations to bring improvements to debug information. Thanks to Jan for pointing the way to a solution for a problem that had not become apparent yet, and for his reviews and suggestions for this paper.

References

[1] Free Standards Group. DWARF Debugging Information Format, Version 4, June 2010. `http://dwarfstd.org/doc/DWARF4.pdf`.

[2] Alexandre Oliva. A plan to fix local variable debug information in gcc. In *Proceedings of the GCC Developers' Summit*, pages 67–76, Ottawa, Ontario, Canada, June 2008. `http://http://www.gccsummit.org/2008/gcc-2008-proceedings.pdf`.