Reprinted from the

Proceedings of the GCC Developers' Summit

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

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Google*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

A Plan to Fix Local Variable Debug Information in GCC

Alexandre Oliva
Red Hat
aoliva@redhat.com

Abstract

As GCC gained more and more optimization passes, the lack of infrastructure to retain a correspondence between source-level and run-time constructs has become a more serious problem. In spite of the growing need for debugging and monitoring optimized code, this has become more and more difficult, because GCC fails to emit location information for so many local user variables, and emits locations that make it seem like variables hold unexpected values at certain points in the program.

This article presents a plan to address these problems, based on annotations introduced early in compilation, in such a way that optimization passes, at little additional effort, keep them accurate and as complete as reasonable all the way to the end of compilation.

1 Introduction

The DWARF Debugging Information Format [3] determines the ways a compiler can communicate the location of user variables at run time to debug-information consumers such as debuggers, program analysis tools, run-time monitors, etc.

One possibility is that the location of a variable is fixed throughout the execution of a function. This is generally good enough for unoptimized programs.

However, for optimized programs, the location of a variable can vary. The variable may be live at some parts of a function, even in multiple locations simultaneously. At other parts, it may be completely unavailable. At others, its value may not be available at any run-time location, but it may still be computable out of constants and other values available at run-time.

DWARF encoding permits the use of location lists for varying locations: tuples with possibly-overlapping ranges of instructions, and location expressions that determine the locations (or values, as in accepted [2] and proposed [1, 4] extensions) of the variable within each range.

Historically, GCC started with the simpler, fixed-location model. In fact, back then, widely-used debug information formats couldn't represent anything better than this.

More recently, GCC gained code to keep track of varying locations, and to emit DWARF debug information accordingly. Unfortunately, very many optimization passes discard information that would be necessary to emit correct and complete variable location lists.

Coalescing, scalarizing, substituting, propagating, and many other transformations prevent the late-running variable tracker from doing a complete or even accurate job. By the time it runs, many variables no longer show up in the retained annotations, although they're still conceptually available.

The variable tracker can't handle sharing of a location by multiple user variables, multiple active locations for the same variable, and it can't tell when a variable is overwritten, if the assignment is optimized away. This last limitation is inherent to a model based on inspecting only actual code, and trying to make inferences from that. In order to be able to represent not only what remained in the code, but also what was optimized away, combined, or otherwise apparently removed, additional information needs to be retained.

This paper describes an approach to maintain this information, as follows. In Section 2, we discuss reasons to improve GCC and compilers in general in this regard. Section 3 sets the goals for the ongoing work. Section 4 proposes extensions to GCC's internal representations so as to retain the additional information needed for better debug information, and Section 5 discusses how to extract better debug information from this

additional information. Section 6 discusses how the proposed approach deals with optimizations that reorder code. Section 7 discusses the stability and testability concerns taken into account when devising this plan, whereas Section 8 discusses other worries expressed in early evaluations of this proposal. Examples of annotations retained in small optimized programs are given in Section 9, followed by a summary of the conclusions in Section 10.

2 Motivation

Debug information was initially meant for debuggers, to enable interactive debugging sessions to locate errors in programs by stopping execution at key points and inspecting internal state without having to go through a recompile, relink, restart cycle so as to add the desired output at the relevant points.

Nevertheless, debugging often involved compiling the program without optimization, because debug information wasn't rich enough to map source constructs to executable constructs in optimized programs.

Nowadays, programs are often big enough that rebuilding them without optimization in order to debug them would take weeks, not counting the time to duplicate the software, hardware, network and timing conditions that led to the error in the first place.

On top of that, information about errors is often available in the form of core files generated by the optimized programs used in production. It is essential to be able to extract information about the error from these files, at the very least to guide the attempts to duplicate the problem in controlled scenarios until it is fully understood. Poor debug information in optimized programs reduces tremendously the usefulness of such core files.

Add to this the growing use of dynamic monitors that rely on debug information to dynamically introduce the evaluation of predicates in programs at run time, to detect error conditions and compliance with specifications or regulations, to collect and log useful information, and even to aid debugging and understanding of complex programs, their behavior, and their performance.

Some of these monitors are essentially scripted interactions with preexisting symbolic debuggers, using whatever interfaces such debuggers provide. Other monitors rely on far lower-level interception and inspection machinery, having to perform the mapping from source-level symbols to run-time locations themselves.

In all of these cases, the mapping between symbols and locations is provided by debug information. If debug information is incomplete, these tools may fail to find the values they were asked to monitor.

Even worse is the case of incorrect debug information, because then the tools may appear to be functioning properly, but operating on incorrect values, thus missing situations they were supposed to catch, generating inaccurate logs, and otherwise failing to abide by specifications and regulations. Buggy debug information causes bugs.

3 Goals

While human users of interactive debuggers can often tolerate and compensate for incompleteness and incorrectness of debug information, automated monitoring tools aren't normally equipped with human adaptability, nor our ability to accumulate experience for detection and tolerance for errors. Therefore, in this work we have set some goals that won't just make debug information more useful for humans, but also usable in such automated tools.

3.1 Correctness

Ensure that, for every user variable for which we emit debug information, the information is correct, i.e., if it provides location or value expressions for a variable in a certain range of instructions, then, for all instructions in that range, the values specified in the debug information must correspond to the value the user variable is bound to.

We say a variable is bound to a value when control flow crosses a theoretical instruction placed at the point of the program in which the user variable is, or should be have been, assigned that value. This theoretical instruction is maintained roughly in place regardless of optimizations that move, remove, or otherwise optimize any code generated to implement the source-level variable modification. This is further detailed in Section 6.

3.2 Completeness

Try to ensure that, for every user variable, at any given point in the program,

- if the variable is live at any location (possibly more than one), all such locations are noted in debug information as locations for the variable at that point;
- otherwise, if the variable is bound to a known constant at that point, the value of the constant is noted in debug information as the value of the variable;
- otherwise, if the variable is bound to a value that, at that point, is computable from other available constants and values of available locations, at least one expression that computes the value is noted in debug information as the value of the variable;

Note that this is not related with the theoretical maximum coverage afforded by an unoptimized program, that we dub theoretical completeness. Some compiler transformations make it impossible for certain variables to be represented in debug information.

Although one could argue that this makes debug information incomplete, if a variable is completely removed from the program, then removing it from debug information is necessary for correctness, and the absence does not fail the completeness criterion because the variable no longer matches any of the cases above.

These criteria have to do with representing source-level concepts that are present in the optimized program, and not representing those that are not. They are not about representing source-level concepts that are optimized away or transformed so profoundly that they can no longer be represented.

3.3 Run-time efficiency vs. debuggability

Debug information is supposed to *represent* the result of optimizations, not *guide* it. When a user requests the compiler to perform certain optimizations, they should be applied to the greatest possible extent, regardless of whether debug information is being generated, of whether performing them would make for poorer debug information.

This is not to say that there shouldn't be options that limit the effect of certain transformations so as to get richer debug information, but rather that these are options that control optimization, not debug information. Options that control the generation of debug information must not prevent optimizations, harm run-time efficiency, or modify the executable code in any way.

3.4 Compile-time efficiency

This proposal strives to avoid using additional memory and CPU cycles that would be needed only to generate debug information, when compiling without generating debug information.

A secondary goal is to minimize the memory and CPU overhead incurred when generating debug information, but this is often at odds with the previous paragraph. This is further discussed in Section 8.

4 Internal Representation

For historical reasons, GCC has two significantly different, even if nearly isomorphic, internal representations: Trees and RTL. This decision has required a lot of code to be duplicated for low-level manipulation and simplification of each of these representations.

Since tracking variables and their values must start early and be carried throughout the complete optimization process, to ensure correctness, it might seem tempting to introduce yet another representation for debug information, decaying both isomorphic representations into a single debug information representation. The drawbacks would be additional duplication of internal representation manipulation code, and the possibility of increasing memory use out of the need for representing information in yet another format.

Another concern is that even the simplest compiler transformations may need to be reflected in debug information. This might indicate a need for modifying every point of transformation in every optimization pass so as to propagate information into the debug information representation. This is undesirable, because it would be very intrusive.

But then, keeping references to the correct values, expressions, or variables as transformations are made is

precisely what optimization passes have to do to perform their jobs correctly. Finding a way to take advantage of this is a very non-intrusive way of keeping debug information accurate. In fact, most transformations wouldn't need any changes whatsoever: uses of variables in debug information can, in most optimization passes, be handled just like any other uses.

Once this is established, a possible representation becomes almost obvious: statements (in Trees) or instructions (in RTL) that assert, to the variable tracker, that a user variable is represented by a given expression, or that bind a user variable to a value:

```
# DEBUG var => expr
```

By `var`, we mean a Tree expression that denotes a user variable, for now. We envision trivially extending it to support components of variables in the future.

By `expr`, we mean a Tree or RTL expression that computes the value of the variable at the point in which the statement or instruction appears in the program, and that the variable is expected to hold until (i) execution crosses another such annotation for that variable, or (ii) the value becomes no longer computable, because all locations containing it or usable to compute it are no longer provably usable to compute it. For example, if the variable is bound to the value of a certain hardware register, and the register is subsequently modified, but the bound value is not known to be available elsewhere, then the variable is regarded as unavailable at that point.

A special value needs to be specified, for each debug annotation representation, that denotes an unavailable variable. Although in some cases this condition can be detected implicitly, as described above, in others we must be able to describe that, at the point of the binding, the value that should be bound to the variable is not available, for example, because it was completely optimized away and it's not even computable any more, or because the compiler has been unable to represent or to keep track of the expected value of the variable at that point.

Furthermore, it might be useful to represent the expression as a list of expressions, to establish larger equivalence classes to begin with, and to get better resistance against complete loss of values.

It may also be useful to distinguish lvalues from rvalues in the representation, but for now we're keeping it simpler, to see if we can make do without the additional complexity.

5 Generating debug information

Generating initial annotations when entering SSA is early enough in the translation that the program will still reflect very reliably the original source code. We will only emit such annotations for user variables that are GIMPLE registers, i.e., variables that are present in the source code, that are not addressable, and that hold scalar values. Addressable or non-scalar user variables don't have varying locations, so we don't need these annotations to generate correct debug information for them.

As optimizations transform the code, the initially-trivial mapping between such user variables and implementation locations gets more and more fuzzy. Even when the compiler retains mnemonic names that resemble user variable names for such implementation locations (GIMPLE registers, RTL pseudos, hardware registers, and stack slots), it is important to keep in mind that source and implementation concepts are in different name spaces, and that the implementation locations cannot be assumed to remain associated with the user variables they were initially named after.

The purpose of the annotations is precisely to establish a mapping from user variables to implementation concepts without preventing optimizations. The choice of focusing not so much on locations, but rather on values, is intended to minimize the impact of optimizations on the ability to represent the value a variable holds, which is what debug information consumers are most often interested in. Actual locations are a slightly secondary issue, that we expect to be able to infer from the value binding annotations, but that may require more explicit annotations, as in the lvalue-vs-rvalue discussion above.

After every assignment to user variables that are GIMPLE registers, we emit a `DEBUG` statement intended to preserve, throughout compilation, the information that, at that point, the user variable was bound to the value of that expression. In other words, after putting an assignment such as the following in SSA form, we emit the debug statement below right after it:

```
x_1 = whatever;
# DEBUG x => x_1
```

Likewise, at control flow merge points, for each PHI node associated with a user variable we introduce in the initial SSA representation, we emit an annotation:

```
# x_3 = PHI <x_1(1), x_2(2)>;
# DEBUG x => x_3
```

Then we let Tree optimizers do their jobs. Whenever they rename, renumber, coalesce, combine, or otherwise optimize a variable, they will most likely automatically update debug statements that mention them as well.

In the rare cases in which the presence of such a statement might prevent an optimization, we need to adjust the optimizer code such that the optimization is not prevented. This most often amounts to skipping or otherwise ignoring debug statements. In a few rare cases, additional code might be needed specifically to adjust debug statements.

During conversion to RTL, the debug statements also decay to debug instructions, and the Tree value expressions are trivially converted to RTL. Conceptually, however, it's still the same representation: a binding from user variable to expression. RTL optimizers will most often adjust debug instructions automatically.

The exceptions can be handled often at no cost: the test for whether an element of the instruction stream is an instruction or some kind of note (that never needs updating) is a range test, in its optimized form. By placing the identifier for a debug instruction at one of the limits of this range, testing for ranges that include or exclude debug instructions requires identical code, except for the constants.

Since most code that tests for `INSN_P` and handles instructions can and should match debug instructions as well, in order to keep them up to date, we extend `INSN_P` so as to match debug instructions, and modify the code in the exceptions that need to skip debug instructions, by using an alternate test, with the same meaning as the original definition of `INSN_P`. These simple and non-intrusive changes are relatively common, but still, by far, the exception rather than the rule. As in Tree level, there are transformations that require special handling of debug annotations, but these are even rarer.

When optimizations are completed, including register allocation and scheduling, it is time to take the data collected in debug instructions and emit debug information out of them. Conceptually, the debug instructions represent points of assignment, at which a user variable ought to evaluate to the annotated expression, maintained throughout compilation. However, when the value of a user variable is available at more than one location (think, for example, stack variable temporarily held also in a register), it is important to note it, so that, if a debugging session attempts to modify the variable, all copies are modified.

The idea is to use some mechanism to determine equivalent expressions throughout a function. At debug instructions, we assert that the value of the named variable is in the equivalence class the expression belongs to. As we scan basic blocks forward and find that expressions in an equivalence class are modified, we remove them from the equivalence class, and thus from the list of available locations for the variables that hold that value. When members of an equivalence class are copied, we add the copies to equivalence class. When equivalent expressions are computed, we add them to the equivalence class. At function calls and `volatile asm` statements, we remove non-function-private memory slots from equivalence classes. At function calls, we also remove call-clobbered registers from all equivalence classes. When no live expression remains in the equivalence class that represents a variable, it is understood that its value is no longer available. At basic block confluences, we combine information from the end states of the incoming blocks, forming, combining, or propagating equivalence classes.

When multiple variables are held in the same equivalence class, some care must be taken to determine which locations can be used as modifiable copies of a variable, which hold incidental copies, and which are read-only values. More investigation is needed to design strategies to make this partitioning, so that the end result is accurate debug information.

Given this plan, debug information should come out as complete as possible, save for transformations that require special handling to update debug annotations properly, but that haven't been improved to do so yet.

6 Scheduling and reordering

Optimizing code involves a lot of moving code around. Basic block reordering, loop unrolling, and other forms of code duplication, movement, or removal that affect placement of sequences of instructions (but not so much the instructions to be executed in a given execution path) have no effect on the debug information annotations presented in this article. When moving, duplicating, or removing code along these lines, debug annotations can be regarded just like regular instructions.

Other than that, debug annotations should generally remain in place, serving as guides for what would amount to the natural execution order of the program, regardless of optimizations that reorder instructions, or move instructions out of loops or conditionals.

For example, if we move to an unconditional block a computation that was only to be performed inside a conditional, the debug annotation that binds the variable to the conditionally-computed value should remain in the conditional block, or be made conditional itself. Likewise, if some computation is hoisted out of a loop, the debug annotation should remain in the loop, where the user expects the assignment to take place.

Moving a computation to an earlier point shouldn't require modification in subsequent debug annotations, but moving it to a later point may, especially when the move crosses the annotation. For example, if an assignment instruction, say `x = y`, is moved past the end of a loop, debug annotations that refer to `x` in their expressions probably need to have it replaced with `y`, so that the binding remains with the same value in spite of the assignment move.

Transformations that reorder instructions within a single block, such as instruction scheduling, don't require modification of annotations. Debug annotations should be maintained after the assignments they refer to, if the assignments are still nearby, and this is trivially accomplished through scheduling dependencies. Other than that, debug annotations should generally have high scheduling priority, so that they are kept right after the corresponding assignment, or moved early when an assignment was hoisted out of a loop, but without causing the instructions they depend on to be scheduled differently.

That said, reordering debug annotations may be undesirable and surprising at times. Care must be taken to not schedule too early debug instructions for assignments whose values are optimized away or unrepresentable: if these have no dependencies, they might be moved too early, to the point of making the range of the previous binding an empty range.

7 Testability

Since debug annotations are added early, and, in most cases, maintained up-to-date by the same code that optimizers use to maintain executable code up-to-date, debug annotations are likely to remain accurate throughout compilation.

The risk of this approach is that the annotations get in the way of optimizations, thus causing executable code to vary depending on whether or not debug information is to be generated. The risk of varying code could be removed at the expense of generating and maintaining debug annotations throughout compilation and just throwing them away at the end. This is undesirable, for it would slow down compilation without debug information and waste memory while at that.

Therefore, we've added testing mechanisms to the compiler build machinery to detect cases in which the presence of debug annotations would cause code changes.

The `bootstrap-debug` `Makefile` target, by default, compiles the second bootstrap stage without debug information, and the third bootstrap stage with it, and then compares all object files after stripping them, a process that discards all debug information.

Furthermore, `make bootstrap4-debug`, after a successful `make bootstrap-debug` followed by `make prepare-bootstrap4-debug-lib-g0`, rebuilds all target libraries without debug information, and compares them with the third stage's target libraries, built with debug information.

At the time of this writing, both tests pass on platforms such as `x86_64-linux-gnu`, `i686-linux-gnu`, `ia64-linux-gnu`, and `ppc64-linux-gnu`.

Additional testing mechanisms should be built in, to exercise a wider range of internal GCC behaviors and extensions, for example, by comparing the compiler output with and without debug information while compiling all of its testsuite.

Even if testing mechanisms fail to catch an error, the generation of debug annotations is controlled by a command-line option, so that any code changes caused by it can be easily avoided, at the expense of the quality of the debug information.

Testing for accuracy and completeness of debug information can be best accomplished using a debugging environment. For example, writing programs of increasing complexity, adding functional-call or `asm` probe points to stabilize the internal execution state, and then examining the state of the program at these probe points in a debugger, shall let us know how accurate and how complete variable location information is.

Measuring accuracy is easy: if you ask for the value of a variable, and get a value other than the expected, there's a bug in the compiler. If you get "unavailable," this can still be regarded as accurate, for locations are always optional. However, it might be incomplete. Telling whether the variable was indeed optimized away, or whether the value is available or computable but the information is missing, is a harder problem, but it's not part of the accuracy test, but rather of the completeness test.

The theoretical-completeness score that an unoptimized program could get is quite often unachievable for an optimized version of the same program, not because the compiler is doing a poor job at maintaining debug information, but rather because the compiler is doing a good job at optimizing it, to the point that no possibility remains of computing the value of certain variables at certain points in the program. This should be taken into account when designing completeness tests, such that they measure completeness with regard to what's available in the optimized program, rather than in comparison with theoretical completeness.

8 Concerns

8.1 Memory consumption

Keeping more information around requires more memory. In order to generate correct debug information, more information needs to be retained throughout compilation.

The only way to arrange for debug information to not require additional memory is to waste memory when not generating debug information. But this is probably undesirable, even if it would minimize the risks of debug annotations affecting optimizations and modifying the generated code.

Therefore, the better debug information we want, the more memory overhead we're going to have to tolerate.

Of course at times we can trade memory for efficiency, using representations that are more compact and more computationally expensive, when we can't have both compactness and efficiency.

At other times, we may trade memory for maintainability. For example, instead of emitting annotations as soon as we enter SSA mode, we could emit them on demand, i.e., whenever we deleted, moved, or significantly modified an SSA assignment for which we would have emitted a debug annotation. Additional memory would be needed to mark assignments that should have gained annotations but haven't, and care must be taken to make sure that transformations aren't made without leaving a correct (even if still implied) debug annotation in place. It is not clear that this would save significant memory, for a large fraction of relevant assignments are probably modified or moved anyway, so it might turn out to be a maintainability and performance loss for small memory gains. More investigation is required to determine whether this is indeed the case.

Worst case, a user may trade memory for debug information quality: if the memory use of this scheme turns out to be too high for some scenario, the user can disable debug information annotations through a command-line option, or disable debug information altogether.

8.2 Intrusiveness

Given that nearly all compiler transformations need to be reflected in debug information to keep it accurate, any solution that doesn't take advantage of this fact is bound to require changes all over the compiler.

This applies perhaps not so much for Tree-SSA passes, that are relatively well-behaved and use a narrow API to make transformations, but very clearly so for RTL passes, that very often modify instructions in place. Passes that reuse locations formerly assigned to user variables as unrelated temporaries should be handled with extra care.

Even when we do use the strength of optimizers to maintain debug information up to date, there are exceptions in which detailed knowledge about the transformation taking place enables us to adjust the annotations properly, if possible, or to discard location information for the variable otherwise.

It is just not possible to hope that information can be kept accurate throughout compilation without any effort from optimizers, or even through a trivial API for a debug information generator. A number of the exceptions that require detailed knowledge about the ongoing transformation would be indistinguishable from other common transformations that would have very different effects on debug information. At this point, any expectations of lower intrusiveness by use of such an API vanish.

By letting optimizers do their jobs on debug annotations, and handling exceptions only at the few locations where they are needed, trivially in most such cases, we keep intrusiveness at a minimum.

Of course we could get even lower intrusiveness by accepting errors in debug information, or accepting the generation of different code depending on debug information command-line options. But these options shouldn't be considered seriously.

8.3 Complexity

The annotations are conceptually trivial and they can be immediately handled by optimizers. It is hard to imagine a simpler design that would still enable us to get right cases such as those in the examples below.

Worrying about the representation of debug annotations as statements or instructions, rather than notes, is missing the fact that, most of the time, we do want the annotations to be updated just like statements and instructions, rather than never updated like notes.

Worrying about the representation of debug annotations in-line, rather than an on-the-side representation, is a valid concern, but it's addressed by the testability of the design, and the in-line representation is highly advantageous, not only for using optimizers to keep debug information accurate, but also for doing away with the need for yet another internal representation and all the efforts into keeping it accurate.

8.4 Optimizations

As discussed in Section 3, correct and more complete debugging information isn't supposed to disable optimizations. Outputting debug information or not isn't supposed to make any difference whatsoever as to the executable code produced by the compiler.

We want to ensure that whatever debug information the compiler generates actually matches the executable code, and that it is as complete as viable.

We don't want to disable optimizations so as to preserve variables or code, so that they could be represented in debug information and provide for a debugging experience more like that of code that is not optimized. If debug information disables any optimization, that's a bug that needs fixing.

Optionally disabling optimizations that lower the quality of debug information is a separate feature, and one that may benefit from this work, but that won't be accomplished through this work.

It is worth mentioning that, while testing the implementation of this design, a number of opportunities for optimization that GCC missed were detected and fixed, others were merely detected so far, and at least one artificial optimization limitation, intended to get better debug information, was kept in place. Once the improved infrastructure is in place and in wide use, this kind of limitation could be removed, for the new infrastructure enables the optimization to be applied to its fullest extent.

9 Examples

It is desirable to be able to represent constants and other optimized-away values, rather than stating that variables have values they can no longer have:

```
int x1 (int x) {
  int i;

  i = 2;
  f(i);
  i = x;
  h();
  i = 7;
  g(i);
}
```

Even if variable i is completely optimized away, a debugger can still print the correct values for i if we keep annotations such as:

```
(debug (var_loc i (const_int 2)))
(set (reg arg0) (const_int 2))
(call (mem (symbol_ref f)))
(debug (var_loc i unknown))
(call (mem (symbol_ref h)))
(debug (var_loc i (const_int 7)))
(set (reg arg0) (const_int 7))
(call (mem (symbol_ref g)))
```

In this case, before the call to h, not only the assignment to i was dead, but also the value of the incoming argument x had already been clobbered. If i had been assigned to another constant instead, debug information could easily represent this, through an extension to DWARF version 3 that enables location lists to contain value expressions, in addition to location expressions.

Another example that covers PHI nodes and conditionals:

```
int x2 (int x, int y, int z) {
  int c = z;
  whatever0(c);
  c = x;
  whatever1();
  if (some_condition)
    {
      whatever2();
      c = y;
      whatever3();
    }
  whatever4(c);
}
```

With SSA infrastructure, this program can be optimized to:

```
int x2 (int x, int y, int z) {
  int c;
  # bb 1
  whatever0(z_0(D));
  whatever1();
  if (some_condition) {
    # bb 2
    whatever2();
    whatever3();
  }
  # bb 3
  # c_1 = PHI <x_2(D)(1), y_3(D)(2)>;
  whatever4(c_1);
}
```

Note how, without debug annotations, c is only initialized just before the call to whatever4. At all other points, the value of c would be unavailable to the debugger, possibly even wrong, if prior assignments to c had survived optimization.

If we were to annotate the SSA definitions forward-propagated into c versions as applying to c, we'd end up with all of x_2, y_3, and z_0 applied to c throughout the entire function, in the absence of additional markers.

Now, with the annotations proposed in this paper, what is initially:

```
int x2 (int x, int y, int z) {
  int c;
  # bb 1
  c_4 = z_0(D);
  # DEBUG c => c_4
  whatever0(c_4);
  c_5 = x_2(D);
  # DEBUG c => c_5
  whatever1();
  if (some_condition) {
    # bb 2
    whatever2();
    c_6 = y_3(D);
    # DEBUG c => c_6
    whatever3();
  }

  # bb 3
  # c_1 = PHI <c_5(D)(1), c_6(D)(2)>
  # DEBUG c => c_1
  whatever4(c_1);
}
```

is optimized into:

```
int x2 (int x, int y, int z) {
  int c;
  # bb 1
  # DEBUG c => z_0(D)
  whatever0(z_0(D));
  # DEBUG c => x_2(D)
  whatever1();
  if (some_condition) {
    # bb 2
    whatever2();
    # DEBUG c => y_3(D)
    whatever3();
  }
  # bb 3
  # c_1 = PHI <x_2(D)(1), y_3(D)(2)>;
  # DEBUG c => c_1
  whatever4(c_1);
}
```

and then, at every one of the inspection points, we get the correct value for variable `c`.

10 Conclusion

This design enables a compiler to emit variable location debug information that complies with the DWARF version 3 standard (although it can further benefit from proposed extensions), and that is likely to be as complete as theoretically possible, with an implementation that is conceptually simple, relatively easy to introduce, trivial to test, and easy to maintain in the long run. Not wasting memory or CPU cycles during non-debug compilation is a welcome bonus.

References

[1] John Bishop and Jim Blandy. Calculate value in DWARF expression, April 2008. `http://dwarfstd.org/ShowIssue.php?issue=071227.1`.

[2] Cary Coutant. Constant expressions in location lists, April 2007. `http://dwarfstd.org/ShowIssue.php?issue=070426.1`.

[3] Free Standards Group. DWARF Debugging Information Format, Version 3, December 2005. `http://dwarfstd.org/Dwarf3.pdf`.

[4] Alexandre Oliva. Constant expressions in location lists, December 2007. `http://dwarfstd.org/ShowIssue.php?issue=071227.1.old`.