

Improving debug info for optimized away parameters

Jakub Jelínek
Red Hat, Inc.
jakub@redhat.com

Abstract

In optimized code, even with VTA and its value based tracking, there are still many variables or parameters for which the compiler doesn't or can't emit location information, either at all or in some range of instructions.

This article discusses proposed DWARF extensions which make it possible to emit location information even for parameters that have been optimized out, either where their value is no longer held anywhere in the current function frame and its registers, or when the compiler didn't pass that parameter altogether, and how this has been implemented in GCC. We also discuss some future possible DWARF extensions which will allow improving the variable/parameter location information coverage even further.

1 Introduction

The DWARF Debugging Information Format[2], especially in its latest version, has several features aimed at debugging highly optimized programs. Individual variables and parameters used in the programs can have location information described by multiple, possibly overlapping, ranges, where in each range the value of the variable or parameter is located in some register, stack or some other memory slot, and newly in DWARF version 4 also has value computed by integral DWARF expressions. The value can be also constructed from pieces where each piece is located in some register, memory or can be computed using some expression. Register or memory locations are preferable over other types of locations, because they are mutable and thus allow the user to modify values of variables from the debugger, constants and expression locations can't.

GCC, starting with version 4.5, uses special debug annotations and value based variable tracking[7] to improve the quality and coverage of the DWARF location information.

```
dwarflint --check=locstats \  
  --locstats:tabulate=0.0:10,99 cclplus  
cov%      samples      cumul  
0.0       462200/76%    462200/76%  
0..10     7932/1%       470132/77%  
11..20    6782/1%       476914/78%  
21..30    8727/1%       485641/80%  
31..40    5814/0%       491455/81%  
41..50    5695/0%       497150/82%  
51..60    4712/0%       501862/82%  
61..70    5436/0%       507298/83%  
71..80    7909/1%       515207/85%  
81..90    13326/2%      528533/87%  
91..99    18189/3%      546722/90%  
100       58482/9%          605204/100%
```

Figure 1: `dwarflint` locstats for non-VTA built `cclplus`

Instead of vague claims of debug information quality improvements for the purposes of this article we'll use a recently written feature of the `dwarflint` tool, part of `elfutils`[1], to gather statistics about location information coverage. For each non-artificial variable or formal parameter DIE it computes what percentage from the code section bytes where it is in scope it has non-empty location description. Figure 1 shows output on GCC `cclplus` binary from SVN revision 164441 which has been built with `cc1` from SVN revision 164425 and `-g -O2 -fno-var-tracking-assignments` options on x86-64, figure 2 the same for `-g -O2`, so the differences between those figures are VTA improvements. The 0.0 line shows the percentage of DIEs with no location information whatsoever, the 100 line shows the percentage of DIEs where there is location information in all code section bytes where the variable or parameter is in scope. In real world usage of course users are always interested in whether there is location information for the variable or parameter they actually need and some variables are more important than others, but that is hard to measure.

```
dwarflint --check=locstats \
  --locstats:tabulate=0.0:10,99 cclplus
cov%      samples      cumul
0.0       185779/30%    185779/30%
0..10     11683/1%       197462/32%
11..20    12419/2%       209881/34%
21..30    20102/3%       229983/38%
31..40    12891/2%       242874/40%
41..50    16269/2%       259143/42%
51..60    13859/2%       273002/45%
61..70    14223/2%       287225/47%
71..80    18975/3%       306200/50%
81..90    27019/4%       333219/55%
91..99    79826/13%     413045/68%
100      192161/31%     605206/100%
```

Figure 2: `dwarflint locstats` for VTA built `cclplus`

Despite these improvements, there are still many variables or parameters with missing or incomplete location information in highly optimized code, including the really important ones. Many of these are for parameters, especially on ABIs where parameters are passed in registers. This can be seen in backtraces, many of the parameters are optimized out. The registers in which the parameters were passed often need to be reused for other things, most often for passing values to another call, and if the parameter value isn't needed in the source afterward or is e.g. just used in some expression based on the parameter that is hoisted into a temporary, the value doesn't need to be saved into some stack slot.

When a value of such a parameter is needed during debugging session, often the choice is rebuild the corresponding compilation unit without optimization and start the debugging session again, or, if the parameter is not modified in the current function, put a breakpoint at the first instruction in the function and remember what values have been passed to the parameters, then continue through the function and substitute the remembered values for the parameters.

In many cases there is another option, that the author has used many times in the past. If the parameter isn't modified in the current function, it is possible to go into the caller's frame (e.g. using GDB `up` command) and see whether it is possible to find the value used as the function argument, through disassembling the code around the call or looking at caller's source. If a constant is loaded into the parameter register, or if a value from a

call saved register is copied into it, or if some stack slot value is loaded into it and the stack slot can't be clobbered by the call, or some expression involving such values, then it is possible to evaluate it in the caller's frame, then go back into the current frame and use the computed value there instead of the parameter for which the debugger has no location information.

While the user can do it himself, the compiler knows or can compute most of the things the user need to check manually. This is not something that can be expressed using current DWARF features, so we have proposed a DWARF extensions for this and tried to make it flexible enough to cover also other uses. Some information from the compiler is needed on the callee side (in particular whether a parameter is never modified) and some information is needed on the caller side (e.g. what values have been passed to the parameters and whether the expressions used to compute them don't involve anything that could be clobbered by the call).

2 Callee side information

On the callee side the only information that needs to be passed from the compiler to the debugger is whether a parameter is never modified in the function. That could be expressed by a flag on the `DW_TAG_formal_parameter` DIE. The debugger when evaluating a value of such a parameter could, if it didn't find any location information for the parameter on the current instruction, look at the caller's information.

While that would be very compact, it isn't sufficiently general. It doesn't allow to describe that the parameter has in certain range a value equal to just some expression involving the value passed to it during call, or that another optimized out variable has a value equal to the optimized out parameter. Say in

```
void foo (int a)
{
    int b = a;
    a = a + 4;
    bar (4);
    baz (6); // Evaluate a and b here.
}
```

the parameter is actually modified, still both `a` and `b` variables can be expressed in terms of the value originally passed to the `a` parameter, even when the register, in which it has been passed, has been already clobbered.

The information whether a parameter hasn't been modified in a function also isn't useful just for the case when the debugger can look up the passed value in the caller. It should be similarly useful for the case when the debugger puts a breakpoint at the start of the function, remembers there all the parameter values and then can use those values later on in the function when the parameter has no location information anymore. Doing that in normal interactive debuggers would be perhaps too expensive, having to put a breakpoint at the start of every function, but if the debugger has a way to say that the user is interested in a value of certain function's parameter and restarts the program, it could be gathered. Or debug information consumers which have the whole debugging script ready before starting a debugging session, such as `systemtap`, it is possible to put breakpoints only where strictly needed - when some parameter won't have other location at a spot where it needs to be evaluated.

From the above it is clear that the callee side extension should be something usable in DWARF expressions, both directly in `DW_AT_location` block or in `.debug_loc` ranges. When it is an operator in DWARF expression, the debugger can do arithmetics on that value and when it can be present in location ranges, it is possible to describe the location using the original register used to pass the parameter where it still holds the value, say that the value of the entity is equal to the original passed value in some other range and that the value is unavailable in yet another range of instructions (e.g. because the parameter has been modified in that portion of the function). As it is going to be used quite frequently in location information, it is important that it is sufficiently compact. It needs to specify what parameter's original value is it, as it should be usable also in location information for other variables or parameters. Some possible choices for identifying the parameter include reference to the parameter's DIE, or the register or stack slot in which the parameter has been passed. The latter is usually more compact.

So, we've proposed new special opcode, `DW_OP_GNU_entry_value`, usable in all DWARF expressions, with the exception of unwind information. This opcode has two operands, the first one is `uleb128 length` and the second is `block` of that length, containing either a simple register or DWARF expression. The semantics is that this operation pushes the value the mentioned register had upon entering current function to the DWARF

```
dwarflint --check=locstats \
  --locstats:tabulate=0.0:10,99 cclplus
```

cov%	samples	cumul
0.0	186203/30%	186203/30%
0..10	10002/1%	196205/32%
11..20	10968/1%	207173/34%
21..30	18425/3%	225598/37%
31..40	11319/1%	236917/39%
41..50	14917/2%	251834/41%
51..60	12381/2%	264215/43%
61..70	12720/2%	276935/45%
71..80	17293/2%	294228/48%
81..90	22943/3%	317171/52%
91..99	65917/10%	383088/63%
100	222118/36%	605206/100%

Figure 3: `dwarflint locstats` for `cclplus` built with `DW_OP_GNU_entry_value` support

expression stack, resp. evaluates the DWARF expression as if it had been evaluated upon entering current function and pushes the TOS value to the current stack. The most common usage is for architectures which pass parameters in registers, where it can be expressed by 3 bytes, e.g.:

```
DW_OP_GNU_entry_value<1, DW_OP_reg4>
```

Some more complex operation can be e.g.:

```
DW_OP_GNU_entry_value<3,
                                DW_OP_breg4<16>
                                DW_OP_deref>
```

which computes the value register 4 had upon entering current function, adds 16 to it and pushes the value address sized memory chunk pointed by that address had upon entering current function.

It could be also generalized even more, as expression that needs to be evaluated at arbitrary places in the function instead of just at the beginning of the function, but then it would also need to encode the instruction address at which it needs to be evaluated. As such generalization would probably be only useful to non-interactive debuggers, we've decided not to generalize it that way. If such need arises in the future, it can be always implemented as a new opcode, leaving the entry value opcode in more compact form.

Implementing this extension in GCC wasn't very hard, thanks to value based variable tracking. All that was needed is adding new RTL code and pushing it as yet another CSELIB location for the function parameter's VALUES. Variable tracking then will use these artificial locations as last resort if the VALUE isn't live anywhere else anymore. `dwarf2out.c` then just needs to handle this new RTL expression and emit `DW_OP_GNU_entry_value` for it.

On `cclplus` binary e.g. `.debug_loc` section grows with `DW_OP_GNU_entry_value` from 43.69MB to 46.26MB and the section contains 113975 `DW_OP_GNU_entry_value` opcodes. Figure 3 shows statistics for the same GCC SVN revision `cclplus`, just built with patched GCC. The numbers are slightly misleading though, because just the fact that `DW_OP_GNU_entry_value` could be emitted as location doesn't necessarily mean the debugger will be able to evaluate its value, that depends also on whether there is value for the parameter in the caller information and whether the debugger can safely use it.

3 Call site information

On the caller site the compiler needs to provide some information for each or at least some of the call sites. For each such call site, at least the address of the call-like instruction is needed and DWARF expressions for each of the arguments that has its value reconstructable from objects not clobbered by the call and some way how to identify which argument it is (either a reference to its DIE, or DWARF expression identifying where it is passed). As the address of the call-like instruction it is best to use the address that shows up in the unwind information, the return address, because the debugger will need to look these entries up from that address.

This information could be emitted in a new section, e.g. `.debug_callinfo`, but we chose to put it instead into `.debug_info` section, as that provides greater flexibility. The information can be used by debug info consumers also for static call graph analysis and other purposes, so it is best to make it extensible and make it possible to use `.debug_abbrev` abbreviations to show up what information is actually provided. With e.g. `-g3` the compiler could emit more detailed call site information, while normally it could emit only what is necessarily needed for purposes of looking up values of optimized out parameters. The whole call site

is represented with `DW_TAG_GNU_call_site` DIE, which is a child of the lexical block, inlined subroutine or whole subprogram's DIE in which the call is present in the source, and has `DW_TAG_GNU_call_site_parameter` children which describe each parameter passed at the call site. The children DIEs then have `DW_AT_location` which describes in which location the parameter is passed and `DW_AT_GNU_call_site_value` DWARF expression that can compute the value passed to the argument. The compiler must guarantee that everything used in the DWARF expression is not clobbered by the call, so debuggers can safely evaluate it.

The GCC implementation involved mainly the variable tracking pass and of course DWARF output. The variable tracking pass has to compute VALUES for each of the arguments of each of the `CALL_INSN` before actually invalidating anything in the call, ensure the registers in which parameters are passed will be considered as interesting for tracking and then just leave the value tracking compute the expressions how to compute the arguments right after the call. As the call invalidated all call clobbered locations, the expressions only use objects that aren't clobbered by the call. As the expressions for the parameters are always emitted in the call site parameter DIEs, it is desirable to turn off expression caching though, to always choose the best suitable expression. The expressions for the parameters are then stored in special new RTL notes, similar to `NOTE_VAR_INSN_LOCATION`. Without value based tracking this would be much harder. In DWARF output the changes are to gather the expressions and other information like the code label from the notes and also allow mapping the `CALL_INSN` back to a `BLOCK` containing the call and later on map that to a lexical block, inlined subroutine or subprogram DIE and emit this as new DIEs and their attributes into the debug information.

3.1 Tail call issues

When evaluating `DW_OP_GNU_entry_value`, it is unfortunately not enough to look up the `DW_TAG_GNU_call_site` DIE corresponding to the caller's return address from the unwind information and evaluating the corresponding argument's expression in the context of the caller. If a call site in function *A* calls function *B* which uses a tail call to the current function *C*, the call

site DIE describes arguments passed to a different function. The tail call doesn't keep any traces in the backtrace, in the unwind information it seems that *C*'s caller is *A* instead of *B*, so if the call site DIE is blindly used without any extra checks, it would use values passed to *B* arguments for *C* arguments. For direct calls this can be solved by adding a `DW_AT_abstract_origin` reference to the called function's DIE which the debugger can compare, for indirect calls there is `DW_AT_GNU_call_site_target` DWARF expression which can be evaluated and compared to the current function's address.

Another issue is that the current function might tail call itself, either directly or indirectly through tail calling other function that eventually tail call back to the current function. In this case using the call site parameter expressions is not reliable, it might describe arguments passed to a call to the current function that then tail called it with different arguments. While detecting that for direct tail calls is possible by the compiler, those are often implemented by tail recursion and thus not interesting for these purposes, in general calls in other compilation units could be involved and thus it can't be detected statically. The checking of this is left to the debugger then, which can do a conservative dynamic check whether the current function might tail call itself. For the dynamic check the debugger needs a guarantee from the compiler that for all tail call sites in the current subprogram `DW_TAG_GNU_call_site` DIEs have been emitted (this is signaled by the `DW_AT_GNU_all_tail_call_sites` flag on the subprogram DIE) and a flag on the call site DIEs whether the call is a tail call (`DW_AT_GNU_tail_call`). If there are no tail call sites in the current function, it can't tail call itself, if there are any indirect tail calls, it needs to conservatively assume it might, otherwise it can recurse into all the subprogram DIEs referenced in the tail call sites and check whether they might tail call the current subprogram.

3.2 Parameters passed by reference

In Fortran most parameters are passed by reference and in other languages like C++ parameters can have reference types. The value of the passed parameter is then usually expressible in `DW_AT_GNU_call_site_value`, most often it is just some stack slot address. It is usually more interesting to know what

value was in that stack slot at the start of the function though, and that value could be changed within the callee. `DW_OP_GNU_entry_value` allows expressing it by dereferencing the parameter register or memory slot value. On the call site side this could be either expressed by adding fake arguments with matching locations, but it is nicer if `DW_TAG_GNU_call_site_parameter` DIEs always correspond to real arguments. We are proposing `DW_AT_GNU_call_site_data_value` attribute containing DWARF extension for this instead.

3.3 Other possible uses of the call site information

The call site DIEs should be usable even for other purposes than just looking up `DW_OP_GNU_entry_value`:

- **Static call graph analysis.** The call site and call site parameter DIEs can have additional attributes, e.g. for indirect method calls reference to the field containing method that is being called, reference to the type of the call and for parameters references to `DW_TAG_formal_parameter` DIEs that can help various tools to static call graph analysis.
- **Enhanced value printing in backtraces.** Currently debuggers print in backtraces the current values of the parameters, including often `<optimized away>` style information. It is often more interesting to know the value that has been passed to the function though. Consider e.g.

```
void foo (char *p)
{
    /* some code */
    p = strchr (p, 0);
    bar ();
    /* some further code */
}
```

If backtrace is printed while in the `bar` call, it will show that `p` has value `""`, which isn't really useful. By looking at the call site information it is often possible to print either the current value, or the value that has been originally passed to the function, or both. To avoid confusion, the debugger in the backtrace should use some sign to make it easier to understand what value is the current

one, what is the original one, or, if both values are known and equal that the only value printed is both the originally passed and current one, and make it perhaps configurable what information should be printed. This should make automatically gathered backtraces by services like `abrt` more useful.

- **Virtual tail call backtraces.** In some cases with the call site information the debugger can reconstruct tail calls in backtraces and even print the arguments passed. If there are tail calls involved, currently debuggers just miss a few frames in the backtrace, which is often confusing to the users who can't understand how can some function appear as a caller of another function when it doesn't directly call it. By looking at the call site DIEs with the `DW_AT_GNU_tail_call` and what function they call it is sometimes possible to find unambiguous tail call path how the caller from the backtrace can tail call the callee, or perhaps just one or more of the initial tail call frames beneath the caller's frame.

4 Completely optimized away parameters

Recent versions of GCC don't pass some arguments at all, if some function binds locally and it is possible to modify all callers, and either some parameter is never used, or an invariant is passed to it. While in the case of invariant argument even current DWARF can express this, the `DW_TAG_formal_parameter` DIE can have `DW_AT_const_value` attribute, if a parameter is not passed because it is never used means currently no location at all for the parameter. `DW_OP_GNU_entry_value` is not usable in this case, as there is no register or memory to which it could refer. Short example:

```
__attribute__((noinline)) static int
foo (int x, int y, int z)
{
    return x + z;
}
int bar (int x, int y, int z)
{
    return foo (x, y - 4, z)
           + foo (x + 8, y, z - 8);
}
```

For this case a different DWARF expression opcode could be used instead, e.g. `DW_OP_GNU_`

`parameter_ref` with DIE reference operand that would reference the `DW_TAG_formal_parameter` DIE of the completely optimized away parameter. In the call site's `DW_TAG_GNU_call_site_parameter` would not have `DW_AT_location` attribute, since it is not passed at all, but instead would have `DW_AT_abstract_origin` referencing the formal parameter DIE.

This hasn't been finalized yet, nor implemented in GCC, but roughly during IPA parameter removal a special debug stmt would need to be added and during expansion a new RTX to hold the special value of the optimized away parameter, which would eventually be emitted as `DW_OP_GNU_parameter_ref`.

5 Floating point expressions

The use of the DWARF expression stack for entry value has issues for floating point values or types larger than target's address, like `long~long` on 32-bit architectures. The DWARF expression stack has been initially designed only for computing addresses of objects and serves very well for that purpose. While in theory it is possible to describe in DWARF version 4 that an optimized out `float` variable has value equal to other variable times 2.5 using `DW_OP_call4` to a DWARF procedure which performs IEEE 754 single multiplication and e.g. for `double` on 32-bit architectures where that type is larger than target's address using `DW_OP_piece` and separate routines to compute the upper and lower half of the IEEE 754 double multiplication, such DWARF procedures would be likely very large, and, especially due to the need of separate routines for different pieces whenever the type is bigger than target's address and varying address size between architectures, hard to maintain. Most of the debuggers already handle floating point arithmetics, so duplicating that support in DWARF procedures is undesirable.

There are various ways how the DWARF expression stack could be extended to handle even floating point values, decimal floating point, fixed point, vectors and integral values larger than target's address in a backwards compatible way.

To address representation of values larger than target's address either there could be an opcode alternative to `DW_OP_stack_value` that wouldn't pop just one, but 2, 4, 8, 16 etc. words from the DWARF stack and

```

#0 ggc_internal_alloc_stat (size=<value optimized out>) at ../../gcc/ggc-page.c:1152
#1 0x0000000000833ba6 in ggc_internal_cleared_alloc_stat (size=40)
   at ../../gcc/ggc-common.c:205
#2 0x0000000000b44272 in ggc_internal_zone_cleared_alloc_stat (s=40,
   z=<value optimized out>) at ../../gcc/ggc.h:316
#3 ggc_alloc_zone_cleared_tree_node_stat (s=40, z=<value optimized out>)
   at ../../gcc/ggc.h:346
#4 make_tree_vec_stat (len=<value optimized out>) at ../../gcc/tree.c:1641
#5 0x0000000000b4d37b in build_int_cst_wide (type=0x7ffff17a0738,
   low=<value optimized out>, hi=<value optimized out>) at ../../gcc/tree.c:1231
#6 0x00000000008537bf in gimplify_expr (expr_p=0x7ffff1784dc0,
   pre_p=0x7fffffdcb0, post_p=0x7fffffda48,
   gimple_test_f=0x849a40 <is_gimple_reg_rhs_or_call>, fallback=1)
   at ../../gcc/gimplify.c:6814

```

Figure 4: Backtrace in cclplus built without DW_OP_GNU_entry_value support

```

#0 ggc_internal_alloc_stat (size=40) at ../../gcc/ggc-page.c:1152
#1 0x0000000000833ba6 in ggc_internal_cleared_alloc_stat (size=40)
   at ../../gcc/ggc-common.c:205
#2 0x0000000000b44272 in ggc_internal_zone_cleared_alloc_stat (s=40,
   z=<value optimized out>) at ../../gcc/ggc.h:316
#3 ggc_alloc_zone_cleared_tree_node_stat (s=40, z=<value optimized out>)
   at ../../gcc/ggc.h:346
#4 make_tree_vec_stat (len=1) at ../../gcc/tree.c:1641
#5 0x0000000000b4d37b in build_int_cst_wide (type=0x7ffff17a0738, low=0, hi=0)
   at ../../gcc/tree.c:1231
#6 0x0000000000b4d9f7 in build_int_cst (type=<value optimized out>, low=0)
   at ../../gcc/tree.c:1039
#7 0x00000000008537bf in gimplify_expr (expr_p=0x7ffff1784dc0,
   pre_p=0x7fffffdcb0, post_p=0x7fffffda48,
   gimple_test_f=0x849ab0 <is_gimple_reg_rhs_or_call>, fallback=1)
   at ../../gcc/gimplify.c:6814

```

Figure 5: Backtrace in cclplus built with DW_OP_GNU_entry_value support

use those as the implicit location's value, or the DWARF stack internal representation could be changed from a stack of address sized integers to a stack of pairs of value type id and a *union* of various types of objects, like the current address sized integer, twice as big integer, IEEE754 single, IEEE754 double, IEEE754 extended, IEEE754 quad, various decimal, fixed and vector formats. DW_OP_stack_value would then pop whatever is the current value and its type from the top of the stack.

Before addressing floating point etc. operations, it is important to mention that the DWARF expression opcode space is small, the opcodes are 8 bit and only roughly 64 opcodes are left in the standard range and even fewer in the vendor space. Thus having separate

opcodes for each of the usual arithmetic operation for the 4 or more different floating point formats, some large integral types, fixed and decimal floating point formats and various vector types would quickly use up all the remaining free opcodes or more. The other format opcodes could thus be either implemented using multiplexing opcodes, like DW_OP_GNU_ieee754_single whose option would be DWARF opcode of the operation, like DW_OP_mul or some special values that would allow conversion in between the formats, or using standard opcodes based on the current value type in the *union* approach of the stack representation. There would be just new opcodes for conversion and for reinterpretation of the raw value data as different format. If operands had different value types, either such combination would result in an error, or one of them would

be first converted to the other type. For each operation it would be specified given the types of arguments what type would the result have, usually the same as the operands, with the exception of comparisons and various conversions or reinterpretation operations.

In the design of such extensions it would be good to also consider whether there should be a way to describe the use of an inferior call in the expression (either limited to const calls like `sqrt`, or arbitrary calls) and if it would be possible to describe calling C++ constructors/destructors for describing e.g. `DW_AT_default_value`.

6 Conclusion

We've proposed DWARF extensions to improve quality of debugging information, both as a GNU vendor extension and submitted it as extension proposals for the upcoming DWARF 5 standard[4][5]. It has been successfully implemented in GCC[3] and GDB[6].

To measure how what percentage of `DW_OP_GNU_entry_value` opcodes will lead to successful recovery of the optimized out value, we run a modified version of GDB on `cc1plus` binary on a smaller testcase, put a breakpoint on `cp_write_global_declarations` and after reaching that breakpoint, executed million times `bt` command followed by `step` command. The modified GDB version gathered statistics on how many `DW_OP_GNU_entry_value` lookups were done while printing the backtraces and how many of them resulted in a non-optimized away value being printed. For the million backtraces there were 2194055 entry value lookups, out of which 960286, or 43.8%, lookups resulted in successful value retrieval. On the same binary `.debug_info` section size grew from 23.19MB to 34.53MB, mainly because of the added call site and call site parameter DIEs and `.debug_loc` section grew from 43.69MB to 46.26MB.

Figure 4 shows an example of a backtrace from `cc1plus` built without entry value support, figure 5 corresponding backtrace with entry value support. In the second figure there are several parameters containing correct values compared to `<value optimized out>` in the first figure and additionally a tail call in the backtrace.

References

- [1] Ulrich Drepper, Roland McGrath, and Petr Machata. Elfutils. <http://git.fedorahosted.org/git/?p=elfutils.git;h=refs/heads/dwarf>.
- [2] Free Standards Group. DWARF Debugging Information Format, Version 4, June 2010. <http://dwarfstd.org/doc/DWARF4.pdf>.
- [3] Jakub Jelínek. GCC entry value and call site support. <http://gcc.gnu.org/ml/gcc-patches/2010-08/msg01475.html>.
- [4] Jakub Jelínek and Roland McGrath. DWARF DW_OP_entry_value extension proposal. <http://dwarfstd.org/ShowIssue.php?issue=100909.1>.
- [5] Jakub Jelínek, Roland McGrath, Jan Kratochvíl, and Alexandre Oliva. DWARF DW_TAG_call_site extension proposal. <http://dwarfstd.org/ShowIssue.php?issue=100909.2>.
- [6] Jan Kratochvíl and the GDB team. GDB entryval support. <http://sources.redhat.com/git/?p=archer.git;h=refs/heads/archer-jankratochvil-entryval>.
- [7] 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://www.gccsummit.org/2008/gcc-2008-proceedings.pdf>.