Cheap generation of debugging information

Xavier Leroy

INRIA Paris

Journées Compilation, 2016-09-07

With thanks to...

Mark Shinwell (Jane Street), for suggesting the use of reachability analysis.

Bernhard Schommer (AbsInt), for implementing most of this stuff in CompCert.

What does it take to do this?

```
(gdb) break unix.c:351
(gdb) run ../boot/ocamlc -v
(gdb) backtrace
#0  caml_executable_name (name=name@entry=0x64f300 <proc_self_ex
    name_len=name_len@entry=256) at unix.c:351
#1  0x000000000042ad2f in caml_main (argv=0x7fffffffdad8) at sta
#2  0x000000000040c65c in main (argc=<optimized out>, argv=<opti
    at main.c:35
(gdb) print name_len
$1 = 256
```

The compiler must have produced copious information for the debugger, and stored it in the object and executable files.

The DWARF file format

DWARF ("Debugging With Arbitrary Record Format") is a standard data format that a compiler can use to describe aspects of a program relevant to a debugger:

- types (predefined, user-defined)
- variables (types, scopes, locations at run-time)
- functions (machine instructions, prologues, epilogues)
- compilation units / modules
- line number table (code address $\leftrightarrow$ source location)
- call frame information
- macro definitions (!)

Compactness

In memory: XML-style tree of tagged records.

On disk (inside ELF object and executable files): compact binary encodings.

In particular, the line number table (which can be very large) is represented as a program for a bytecode machine whose execution (by the debugger) reconstructs the table.

Address	File	Line	Col	Stmt	Block	End	Prolog	Epilog	ISA
0x0	0	42	0	yes	no	no	no	no	0
0x9	0	44	0	yes	no	no	no	no	0
0x1a	0	44	0	yes	no	no	no	no	0
0x24	0	46	0	yes	no	no	no	no	0
0x2c	0	47	0	yes	no	no	no	no	0
0x32	0	49	0	yes	no	no	no	no	0
0x41	0	50	0	yes	no	no	no	no	0
0x47	0	51	0	yes	no	no	no	no	0
0x50	0	53	0	yes	no	no	no	no	0
0x59	0	54	0	yes	no	no	no	no	0
0x6a	0	54	0	yes	no	no	no	no	0
0x73	0	55	0	yes	no	no	no	no	0
0x7b	0	56	0	yes	no	yes	no	no	0

File 0: `strndup.c`
File 1: `stddef.h`

Expressiveness

Naively, the run-time location of a variable is one of:

- an absolute memory address
- a stack-relative memory address
- a register

To handle more complicated situations (e.g. displays, closures, pass-by-reference, etc), DWARF describes the location of a variable by a program for a stack machine, which can

- access registers (incl. SP) and symbol table;
- perform pointer arithmetic;
- dereference pointers;
- test and branch;
- recombine data that's been split up (e.g. high 32 bits here, low 32 bits there).

Outline

1. The DWARF file format
2. Adding DWARF generation to a compiler
3. Warm up: line number information
4. Black belt: locations of local variables
5. In closing...

Demanding users

Your cute academic compiler is successful.

Industrial users are serious about using it.

Someone is serious about distributing and supporting it as a product.

They all expect a working -g option!

Demanding users

Your cute academic compiler is successful.

Industrial users are serious about using it.

Someone is serious about distributing and supporting it as a product.

They all expect a working -g option!

Time to panic?

Producing DWARF info

Encoding the information in DWARF binary format:

- A lot of code to write
- Not much help from the assembler
- Some but not many libraries available
- Not my problem today.

Producing the information in the first place:

- collect it, esp. from source code
- transport it through the compiler
- preserve it through optimizations (if at all possible).

Producing DWARF info

Encoding the information in DWARF binary format:

- A lot of code to write
- Not much help from the assembler
- Some but not many libraries available
- Not my problem today.

Producing the information in the first place:

- collect it, esp. from source code
- transport it through the compiler
- preserve it through optimizations (if at all possible).

TIME TO PANIC??

How to go about it?

Bypass the compilation chain:

- Generate as much info as possible from the source AST.
- E.g. infos on types and on global variables.
- Call frame info uses only late back-end info.

Instrument the compilation passes:

- E.g. add a "line number info" field to every node of your CFG.
- A lot of work.
- Especially if your pass is formally verified.

Piggy-back on an existing mechanism:

- For CompCert: built-in functions and annotations.
- For GCC and Clang: inline assembly (possibly).

Built-in functions in CompCert

```
x = __builtin_fmin(y, z);
```

Look like normal function calls.

Most compilation passes treat them conservatively, as calls to unknown functions with unknown effects.

At the end of the compilation pipeline, turned into canned sequences of machine instructions (e.g. the `MINSD` instruction of x86-SSE2).

Used heavily in CompCert to deal with annoying C features that should not be optimized, such as volatile memory accesses or inline assembly.

Code annotations in CompCert

```
__builtin_annot("CODE MARKER");
```

A built-in function that generates zero machine instructions, just a comment in the assembly file at the code point where the annotation occurs:

```
; CODE MARKER
```

Used by Airbus to communicate information to a WCET static analyzer. (See ERTS 2012 paper by Ricardo Bedin França et al.)

GCC equivalent:

```
asm volatile ("; CODE MARKER");
```

Outline

The line numbers problem

Given: a C abstract syntax tree with (filename, line number) information for every statement.

Objective: generate `.file` and `.loc` directives in the assembly file.

```
.file 1 "foo.c"
.loc 1 99
```

(The GNU assembler takes care of DWARF-encoding this info.)

Whimsical idea

Rewrite the C source by printing source location before execution of every nontrivial statement.

```
printf("fact.c:10");    x = 10;
printf("fact.c:11");    y = fact(n);
printf("fact.c:12");    printf("factorial %d is %d\n", x, y);
```

We get a kind of self-describing program: at run-time it prints the source locations executed, correctly interspersed with the actual output of the program.

The CompCert approach

Rewrite the C source, putting the source location in a `__builtin_debug_line` before every nontrivial statement.

```
__builtin_debug_line("fact.c:10");   x = 10;
__builtin_debug_line("fact.c:11");   y = fact(n);
__builtin_debug_line("fact.c:12");   printf("factorial %d is %
```

At assembly generation time, `__builtin_debug_line` is turned into appropriate `.loc` and `.file` directives.

Nothing is printed, but the optimizations don't know that, hence they globally preserve the annotations and their placement relative to the rest of the code.

Does it work in practice?

Yes, fairly well, provided that the code linearization heuristic (the part that lays out the CFG as a sequence of asm instructions) keeps basic blocks contiguous:

```
 __builtin_debug_line("fact.c:10");
x = 10;
```

and doesn't do stupid things like moving `__builtin_debug_line` off-line:

```
     goto L1
L2:  x = 10;
     ...                                ...

L1:  __builtin_debug_line("fact.c:10")
     goto L2
```

Outline

Debugging information for local variables

(Local variable = `auto`, non-`static` block-scoped variable in C.)

For a local variable, the debugger needs to know:

- Name (easy)
- Type (easy)
- Scope (set of PCs where we can talk about this variable)
- Live range (set of PCs where the variable has a value)
  (e.g. from first assigment to end of scope)
- DWARF programs to determine the value of the variable at every PC of its live range.

Beware of optimizations

Classic optimizations can change local variable info a lot:

- Shorten the live range.
  (Register allocation reusing a register after last use.)
- Split the live range into unconnected components.
  (SSA conversion, live range splitting).
- Remove the variable and its initializations.
  (Dead code elimination).
- Turn the variable into a constant.
  (Constant propagation)
- Turn the variable into a derived quantity.
  (E.g. `x` is always `y + 1`)
- Assign different locations at different PCs.
  (E.g. a register and a stack slot, with spills and reloads)

How to keep track of all of this?

Code instrumentation to the rescue

Rewrite the C source (more exactly: the Clight intermediate representation), adding a builtin after each assignment to a local variable, tracing the name and new value of the variable.

```
int f(int n)
{
                                __builtin_debug_setvar("n", n);
  int r = n & 1;                __builtin_debug_setvar("r", r);
  n = n + r;                    __builtin_debug_setvar("n", n);
  r = 1;                        __builtin_debug_setvar("r", r);
  while (n > 0) {
    n--;                        __builtin_debug_setvar("n", n);
    r = r * n;                  __builtin_debug_setvar("r", r);
  }
  return r;
}
```

(Hint: what we really want to trace are the beginnings of live ranges for local variables. This is an overapproximation.)

After compilation

After optimizations, register allocation, and CFG linearization, we get the following Linear intermediate code:

```
f() {
        AX = param(0, int)
        __builtin_debug_setvar("n", AX)       // n starts in AX
        CX = AX
        CX = CX & 1
        __builtin_debug_setvar("r", CX)       // r starts in CX
        DX = AX + CX + 0
        __builtin_debug_setvar("n", DX)       // n is now in DX
        AX = 1
        __builtin_debug_setvar("r", AX)       // and r in AX
   11:  CX = DX
        DX = CX + -1
        __builtin_debug_setvar("n", DX)
        if (CX <=s 0) goto 3
        AX = AX * DX
        __builtin_debug_setvar("r", AX)
        goto 11
    3:  return
}
```

Reaching definitions analysis

Just after

```
__builtin_debug_setvar("n", AX)
```

we know that register `AX` holds the current value of variable `n`.

What about other program points? We'd like to have, at every program point, a set of pairs

(source variable name, register or stack location)

recording where we can find the current values of the source variables.

Reaching definitions analysis

$$\{ \text{(source variable name, register or stack location)} \}$$

To compute these sets, CompCert performs a reaching definitions analysis on the Linear code:

- The `__builtin_debug_setvar`(n, r) are the definitions we track. It adds (n, r) to the set and removes any other (n, r').
- Any assignment to r kills all (n, r).
- Moves between registers or stack slots can be tracked more precisely.

Reaching definitions analysis

```
f() {
        AX = param(0, int)
        __builtin_debug_setvar("n", AX)     // (n in AX)
        CX = AX                             // (n in AX)
        CX = CX & 1                         // (n in AX)
        __builtin_debug_setvar("r", CX)     // (n in AX) (r in CX)
        DX = AX + CX + 0                    // (n in AX) (r in CX)
        __builtin_debug_setvar("n", DX)     // (n in DX) (r in CX)
        AX = 1                              // (n in DX) (r in CX)
        __builtin_debug_setvar("r", AX)     // (n in DX) (r in AX)
   11:  CX = DX                             // (n in DX) (r in AX)
        DX = CX + -1                        //           (r in AX)
        __builtin_debug_setvar("n", DX)     // (n in DX) (r in AX)
        if (CX <=s 0) goto 3                // (n in DX) (r in AX)
        AX = AX * DX                        // (n in DX)
        __builtin_debug_setvar("r", AX)     // (n in DX) (r in AX)
        goto 11                             // (n in DX) (r in AX)
    3:  return
}
```

Differential encoding

To help with DWARF encoding, we recode the results of reaching definitions by marking only the beginning and the end of a range of reaching definitions.

```
f() {
        AX = param(0, int)
        __builtin_debug_startrange("n", AX)
        CX = AX
        CX = CX & 1
        __builtin_debug_startrange("r", CX)
        __builtin_debug_endrange("n")
        DX = AX + CX + 0
        __builtin_debug_startrange("n", DX)
        __builtin_debug_endrange("r")
        AX = 1
        __builtin_debug_startrange("r", AX)
   11:  CX = DX
        DX = CX + -1
        if (CX <=s 0) goto 3
        AX = AX * DX
        goto 11
    3:  __builtin_debug_endrange("r")
        __builtin_debug_endrange("n")
        return
```

Assembly code generation

```
f:      subl    $20, %esp
        leal    24(%esp), %edx
        movl    %edx, 0(%esp)
        movl    0(%edx), %eax
LD1:    # STARTRANGE n %eax
        movl    %eax, %ecx
        andl    $1, %ecx
LD2:    # STARTRANGE r %ecx
LD3:    # ENDRANGE n
        leal    0(%eax,%ecx,1), %edx
LD4:    # STARTRANGE n %edx
LD5:    # ENDRANGE r
        movl    $1, %eax
LD6:    # STARTRANGE r %eax
L100:   movl    %edx, %ecx
        leal    -1(%ecx), %edx
        testl   %ecx, %ecx
        jle     L101
        imull   %edx, %eax
        jmp     L100
L101:
LD8:    # ENDRANGE r
LD9:    # ENDRANGE n
        addl    $20, %esp
        ret
```

```
[LD1,LD3]: n is in %eax
[LD4,LD9]: n is in %edx
[LD2,LD5]: r is in %ecx
[LD6,LD8]: r is in %eax
```

Fine points of the approach

Putting a `__builtin_debug_setvar` after each variable assignment or initialization works only for scalar variables whose address is not taken (via the `&` operator).

CompCert preallocates those other local variables (arrays, structs, addressable scalars) in the stack frame, before optimization. They are not subject to register allocation and live range splitting. Hence, a single

```
__builtin_debug_addrvar("t", &t);
```

at the beginning of the scope of `t` suffices to track its location.

Optimizations of stack-allocated variables

```
void f(int x)
{ g(&x);              // forces stack-allocation of x
  x = x + 15;         // x is temporarily lifted to a register
  x = x >> 2;
  g(&x);
}
```

Here, the "store into stacked `x`"; "reload from stacked `x`" sequence between the two assignments to `x` is optimized away, causing `x`'s current value to be temporarily held in a register, and its value in the stack location to be out of date.

We should tell the debugger about this but we don't. GCC doesn't either.

Scoping issues

Sometimes the reachability analysis infers locations beyond the end of the scope of a variable:

```
int f(int x)
{
  int j = x + 1;
  { double i = x + 2; g(i); }
  return j;                  // i still available in a register
}
```

This is harmless and sometimes useful.

Scoping issues

However, if we have several variables with the same name and overlapping scopes, more work is needed to distinguish them correctly:

```
int f(int x)
{
  int i = x + 1;
  { double i = x + 2; g(i); }
  return i;                  // i here refers to "int i" only
}
```

Outline

1. The DWARF file format
2. Adding DWARF generation to a compiler
3. Warm up: line number information
4. Black belt: locations of local variables
5. In closing...

Putting this approach into practice

Available in CompCert since version 2.5 (PowerPC)
and 2.6 (ARM, x86).

Advanced prototype for the OCaml native-code compiler.

How do others proceed?

No idea how other production-quality compilers handle the generation of debugging information.

War stories welcome!