Visualizing the effects of compiler optimizations

Pietro Ghiglio

November 18, 2020

Table of Contents

Table of Contents

Background

- I'm currently working on my thesis about source code-level visualization of the energy consumption of a program, using debug information.
- Compiler optimizations "obfuscate" the mapping between source code locations and assembly instructions.
- I needed to assess their impact on debug information, eventually "filling the gaps" they may have been left by compiler optimizations.

Background - 2

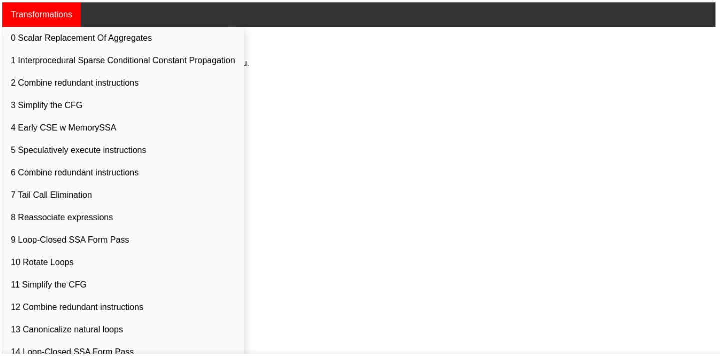
I produced a tool that:

- Offers a view of the modules before and after a transformation pass has run, highlighting the differences.
- Attempts to automatically propagate debug information when instructions are replaced by other instructions during transformation passes.

Table of Contents

Dropdown menu

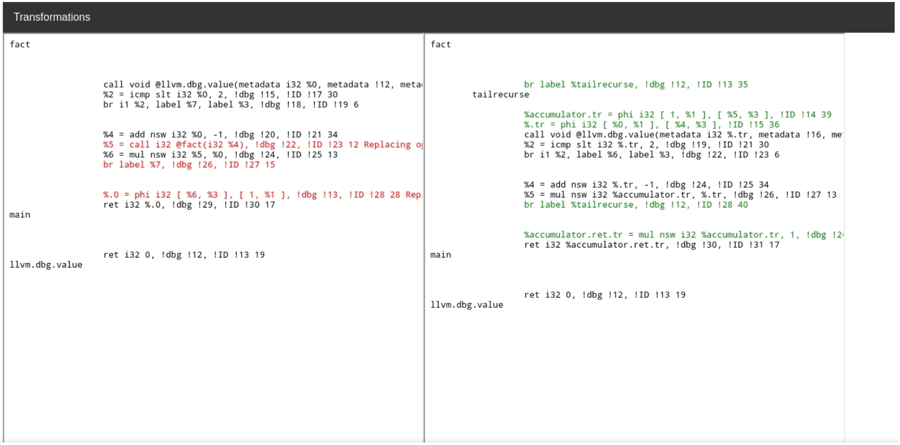
Dropdown menu with the list of transformation passes that opt performed.



0 Scalar Replacement Of Aggregates
1 Interprocedural Sparse Conditional Constant Propagation
2 Combine redundant instructions
3 Simplify the CFG
4 Early CSE w MemorySSA
5 Speculatively execute instructions
6 Combine redundant instructions
7 Tail Call Elimination
8 Reassociate expressions
9 Loop-Closed SSA Form Pass
10 Rotate Loops
11 Simplify the CFG
12 Combine redundant instructions
13 Canonicalize natural loops
14 Loop-Closed SSA Form Pass

Diff view

For each pass, shows the module before and after the pass, highlighting the differences.

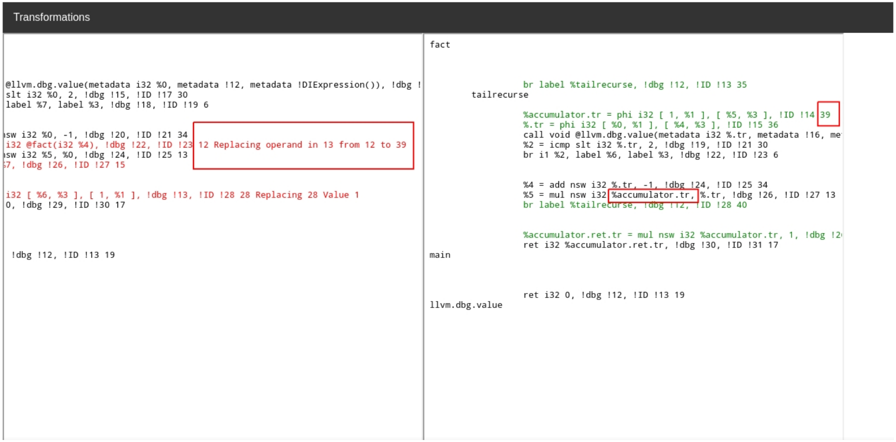


```
fact



                  call void @llvm.dbg.value(metadata i32 %0, metadata !12, meta
                  %2 = icmp slt i32 %0, 2, !dbg !15, !ID !17 30
                  br i1 %2, label %7, label %3, !dbg !18, !ID !19 6


                  %4 = add nsw i32 %0, -1, !dbg !20, !ID !21 34
                  %5 = call i32 @fact(i32 %4), !dbg !22, !ID !23 12 Replacing o
                  %6 = mul nsw i32 %5, %0, !dbg !24, !ID !25 13
                  br label %7, !dbg !26, !ID !27 15


                  %.0 = phi i32 [ %6, %3 ], [ 1, %1 ], !dbg !13, !ID !28 28 Rep
                  ret i32 %.0, !dbg !29, !ID !30 17
main



                  ret i32 0, !dbg !12, !ID !13 19
llvm.dbg.value
```

```
fact



                  br label %tailrecurse, !dbg !12, !ID !13 35
        tailrecurse

                  %accumulator.tr = phi i32 [ 1, %1 ], [ %5, %3 ], !ID !14 39
                  %.tr = phi i32 [ %0, %1 ], [ %4, %3 ], !ID !15 36
                  call void @llvm.dbg.value(metadata i32 %.tr, metadata !16, me
                  %2 = icmp slt i32 %.tr, 2, !dbg !19, !ID !21 30
                  br i1 %2, label %6, label %3, !dbg !22, !ID !23 6


                  %4 = add nsw i32 %.tr, -1, !dbg !24, !ID !25 34
                  %5 = mul nsw i32 %accumulator.tr, %.tr, !dbg !26, !ID !27 13
                  br label %tailrecurse, !dbg !12, !ID !28 40


                  %accumulator.ret.tr = mul nsw i32 %accumulator.tr, 1, !dbg !2
                  ret i32 %accumulator.ret.tr, !dbg !30, !ID !31 17
main



                  ret i32 0, !dbg !12, !ID !13 19
llvm.dbg.value
```

Replaced information

Displays information about the fact that an instruction has been replaced by another instruction, or that an instruction has been moved to a different location in the module.



Transformations

```
@llvm.dbg.value(metadata i32 %0, metadata !12, metadata !DIExpression()), !dbg !
slt i32 %0, 2, !dbg !15, !ID !17 30
label %7, label %3, !dbg !18, !ID !19 6


sw i32 %0, -1, !dbg !20, !ID !21 34
i32 @fact(i32 %4), !dbg !22, !ID !23 12 Replacing operand in 13 from 12 to 39
sw i32 %5, %0, !dbg !24, !ID !25 13
7, !dbg !26, !ID !27 15


i32 [ %6, %3 ], [ 1, %1 ], !dbg !13, !ID !28 28 Replacing 28 Value 1
0, !dbg !29, !ID !30 17



 !dbg !12, !ID !13 19
```

```
fact



         br label %tailrecurse, !dbg !12, !ID !13 35
    tailrecurse

         %accumulator.tr = phi i32 [ 1, %1 ], [ %5, %3 ], !ID !14 39
         %.tr = phi i32 [ %0, %1 ], [ %4, %3 ], !ID !15 36
         call void @llvm.dbg.value(metadata i32 %.tr, metadata !16, me
         %2 = icmp slt i32 %.tr, 2, !dbg !19, !ID !21 30
         br i1 %2, label %6, label %3, !dbg !22, !ID !23 6


         %4 = add nsw i32 %.tr, -1, !dbg !24, !ID !25 34
         %5 = mul nsw i32 %accumulator.tr, %.tr, !dbg !26, !ID !27 13
         br label %tailrecurse, !dbg !12, !ID !28 40


         %accumulator.ret.tr = mul nsw i32 %accumulator.tr, 1, !dbg !2
         ret i32 %accumulator.ret.tr, !dbg !30, !ID !31 17
main



         ret i32 0, !dbg !12, !ID !13 19
llvm.dbg.value
```

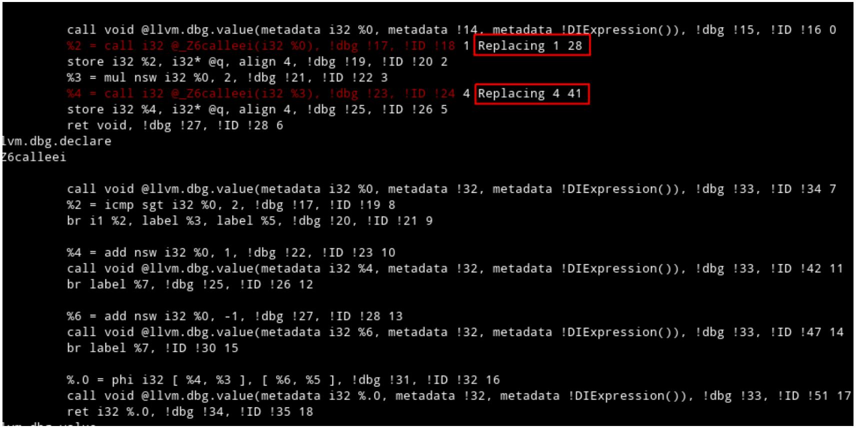
Table of Contents

Debug info propagation

Using the information about whether an instruction has been replaced by another one, we can assign the debug location of the replaced instruction to the replacing one.

Example - Function inlining

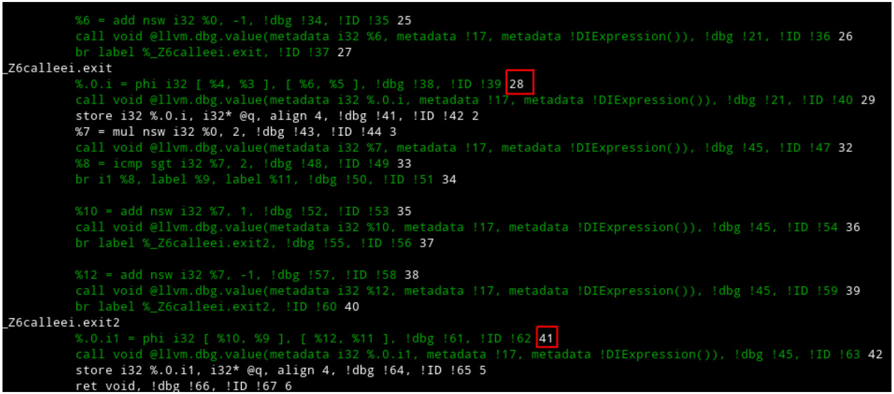
The two call instructions are replaced with the last operations of the inlined functions.



```
        call void @llvm.dbg.value(metadata i32 %0, metadata !14, metadata !DIExpression()), !dbg !15, !ID !16 0
        %2 = call i32 @_Z6calleei(i32 %0), !dbg !17, !ID !18 1 Replacing 1 28
        store i32 %2, i32* @q, align 4, !dbg !19, !ID !20 2
        %3 = mul nsw i32 %0, 2, !dbg !21, !ID !22 3
        %4 = call i32 @_Z6calleei(i32 %3), !dbg !23, !ID !24 4 Replacing 4 41
        store i32 %4, i32* @q, align 4, !dbg !25, !ID !26 5
        ret void, !dbg !27, !ID !28 6
lvm.dbg.declare
Z6calleei

        call void @llvm.dbg.value(metadata i32 %0, metadata !32, metadata !DIExpression()), !dbg !33, !ID !34 7
        %2 = icmp sgt i32 %0, 2, !dbg !17, !ID !19 8
        br i1 %2, label %3, label %5, !dbg !20, !ID !21 9

        %4 = add nsw i32 %0, 1, !dbg !22, !ID !23 10
        call void @llvm.dbg.value(metadata i32 %4, metadata !32, metadata !DIExpression()), !dbg !33, !ID !42 11
        br label %7, !dbg !25, !ID !26 12

        %6 = add nsw i32 %0, -1, !dbg !27, !ID !28 13
        call void @llvm.dbg.value(metadata i32 %6, metadata !32, metadata !DIExpression()), !dbg !33, !ID !47 14
        br label %7, !ID !30 15

        %.0 = phi i32 [ %4, %3 ], [ %6, %5 ], !dbg !31, !ID !32 16
        call void @llvm.dbg.value(metadata i32 %.0, metadata !32, metadata !DIExpression()), !dbg !33, !ID !51 17
        ret i32 %.0, !dbg !34, !ID !35 18
```

Example - Function inlining cont.

The two phi-nodes do not have line/column information, but we can assign them the locations of the old call instructions that they replace.



```
        %6 = add nsw i32 %0, -1, !dbg !34, !ID !35 25
        call void @llvm.dbg.value(metadata i32 %6, metadata !17, metadata !DIExpression()), !dbg !21, !ID !36 26
        br label %_Z6calleei.exit, !ID !37 27
_Z6calleei.exit
        %.0.i = phi i32 [ %4, %3 ], [ %6, %5 ], !dbg !38, !ID !39 28
        call void @llvm.dbg.value(metadata i32 %.0.i, metadata !17, metadata !DIExpression()), !dbg !21, !ID !40 29
        store i32 %.0.i, i32* @q, align 4, !dbg !41, !ID !42 2
        %7 = mul nsw i32 %0, 2, !dbg !43, !ID !44 3
        call void @llvm.dbg.value(metadata i32 %7, metadata !17, metadata !DIExpression()), !dbg !45, !ID !47 32
        %8 = icmp sgt i32 %7, 2, !dbg !48, !ID !49 33
        br i1 %8, label %9, label %11, !dbg !50, !ID !51 34

        %10 = add nsw i32 %7, 1, !dbg !52, !ID !53 35
        call void @llvm.dbg.value(metadata i32 %10, metadata !17, metadata !DIExpression()), !dbg !45, !ID !54 36
        br label %_Z6calleei.exit2, !dbg !55, !ID !56 37

        %12 = add nsw i32 %7, -1, !dbg !57, !ID !58 38
        call void @llvm.dbg.value(metadata i32 %12, metadata !17, metadata !DIExpression()), !dbg !45, !ID !59 39
        br label %_Z6calleei.exit2, !ID !60 40
_Z6calleei.exit2
        %.0.i1 = phi i32 [ %10, %9 ], [ %12, %11 ], !dbg !61, !ID !62 41
        call void @llvm.dbg.value(metadata i32 %.0.i1, metadata !17, metadata !DIExpression()), !dbg !45, !ID !63 42
        store i32 %.0.i1, i32* @q, align 4, !dbg !64, !ID !65 5
        ret void, !dbg !66, !ID !67 6
```

Table of Contents

Workflow

- I added a command line option to opt (-pn) that prints the module after each transformation (like -p) and logs all the operations performed by a pass (instruction creation, deletion, replacing and moving).
- Currently, I redirect this output to a file, and I've written a tool that parses it, propagates debug locations and outputs the html files.
- I'm thinking about integrating the propagation and html files production directly into opt, removing the "redirect and parse" steps, which will speed up the entire process, removing the need of reading and writing to disk.

Implementation

Two (very simple) utility passes:

- -addUnique: adds a unique id the all the instructions in the module.
- -printPassName: simply prints the name of the next pass in the pipeline (used to show pass names in the dropdown menu).

I've modified the LLVM functions for creating, removing, moving and replacing instructions, making them also produce a log entry of the performed operation.
I also had to manually inspect the code of some transformation passes, adding log entries when needed.

Table of Contents

Final remarks

- Showing the changes that a module underwent could have been done by simply "diffing" the string representations of the module. There's some work in that sense by the LLVM community (eg. check out the talk "Understanding Changes made by a Pass in the Opt Pipeline." at the last LLVM conference).
- I needed something with a "deeper understanding" of what the transformation did, in order to automatically propagate debug information.
- This required modifying the code of some of the core LLVM features, making my implementation a bit "heavy" from the point of view of maintenance and, eventually, distribution. But it could very well be a stand-alone tool.