“Great Compiler Challenge”

A first look at GCC

- GCC, born in 1987 with blessings of Richard Stallman, the pioneer of FSF.
- More than 15 million lines of code!
- 300+ optimization passes
- Support for 40+ different architectures upstream.
- Actively maintained front-ends for C, C++, Ada, Go, Fortran and in-progress for Rust. Historically had support for Java.
- Can be used as both AOT and JIT compiler.
- De-facto system compiler on most linux distros.

Outline

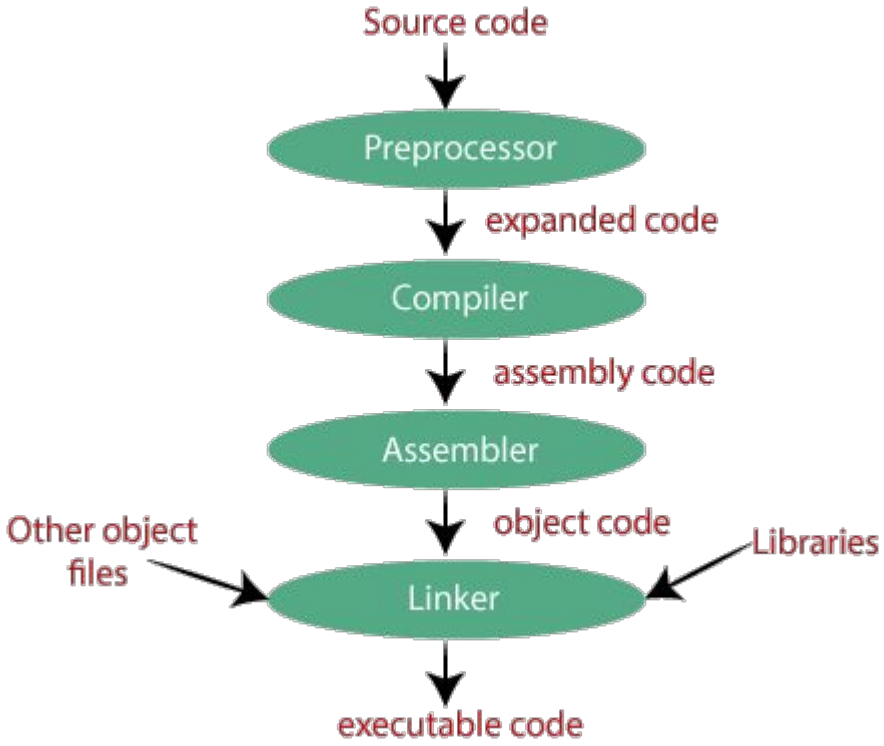
- See how GCC compiles a sample test-case
- What optimizations GCC performs with higher optimization levels like -O2 for the test case
- We will implement a new transform in GCC
- Compare code-gen differences and see how performance compares to -O0

Sample test-case

```c
#include <math.h>

double f(double x)
{
  return (sin (x) * sin (x))
         + (cos (x) * cos (x));
}
```

- What does f return ?
- gcc -O0 sin2cos2.c -lm -save-temps
- What is gcc in above command ?
- Options:
  - -O0: Don’t perform any optimizations
  - -lm: Link math library
  - -save-temps: Save intermediate files

Source to Executable

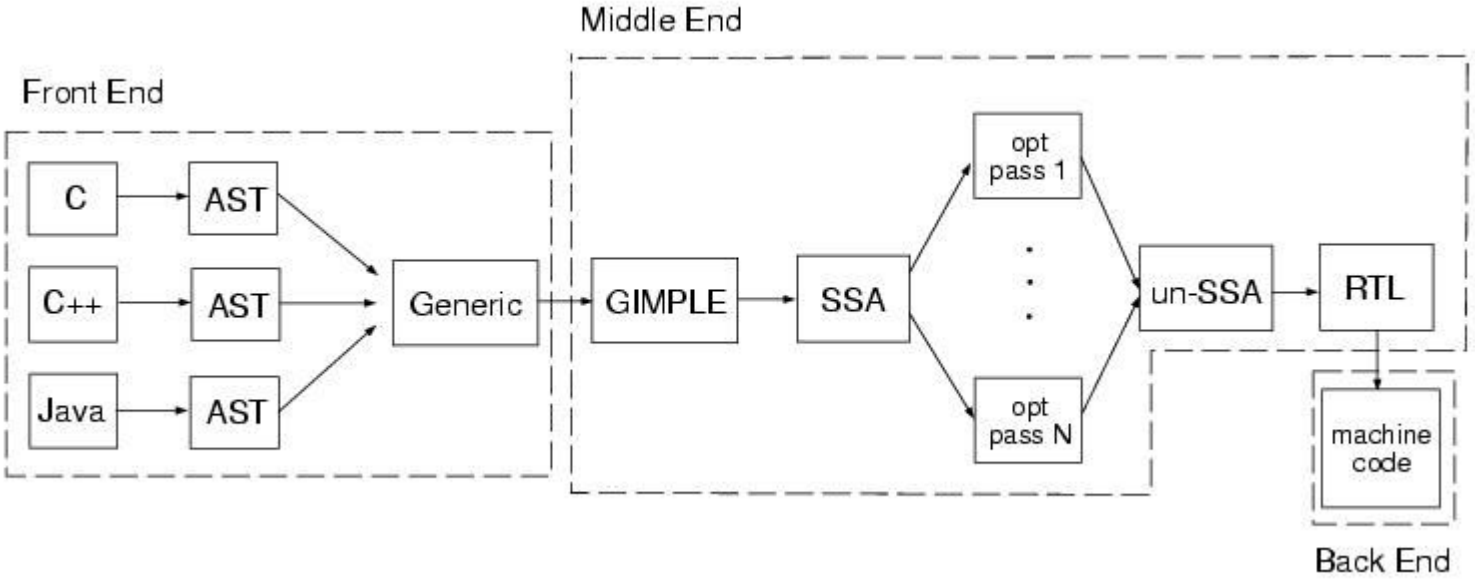


Driver invoking various toolchain components

```
Preprocessor invocation:
/usr/lib/gcc/aarch64-linux-gnu/9/cc1 -E -quiet -v -imultiarch aarch64-linux-gnu sin2cos2.c -mlittle-endian -mabi=lp64 -O2 -fpch-
preprocess -fasynchronous-unwind-tables -fstack-protector-strong -Wformat -Wformat-security -fstack-clash-protection -o sin2cos2.i

Compiler invocation:
 /usr/lib/gcc/aarch64-linux-gnu/9/cc1 -fpreprocessed sin2cos2.i -quiet -dumpbase sin2cos2.c -mlittle-endian -mabi=lp64 -auxbase sin2cos2
-O2 -version -fasynchronous-unwind-tables -fstack-protector-strong -Wformat -Wformat-security -fstack-clash-protection -o sin2cos2.s

Assembler invocation:
 as -v -EL -mabi=lp64 -o sin2cos2.o sin2cos2.s

Linker invocation:
 /usr/lib/gcc/aarch64-linux-gnu/9/collect2 -plugin /usr/lib/gcc/aarch64-linux-gnu/9/liblto_plugin.so -plugin-opt=/usr/lib/gcc/aarch64-
linux-gnu/9/lto-wrapper -plugin-opt=-fresolution=sin2cos2.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-
opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --eh-frame-hdr --hash-style=gnu --as-
needed -dynamic-linker /lib/ld-linux-aarch64.so.1 -X -EL -maarch64linux --fix-cortex-a53-843419 -pie -z now -z relro -o sin2cos2
/usr/lib/gcc/aarch64-linux-gnu/9/../../../aarch64-linux-gnu/Scrt1.o /usr/lib/gcc/aarch64-linux-gnu/9/../../../aarch64-linux-gnu/crti.o
/usr/lib/gcc/aarch64-linux-gnu/9/crtbeginS.o -L/usr/lib/gcc/aarch64-linux-gnu/9 -L/usr/lib/gcc/aarch64-linux-gnu/9/../../../aarch64-
linux-gnu -L/usr/lib/gcc/aarch64-linux-gnu/9/../../../../lib -L/lib/aarch64-linux-gnu -L/lib/../lib -L/usr/lib/aarch64-linux-gnu -
L/usr/lib/../lib -L/usr/lib/gcc/aarch64-linux-gnu/9/../../.. sin2cos2.o -lm -lgcc --push-state --as-needed -lgcc_s --pop-state -lc -lgcc
--push-state --as-needed -lgcc_s --pop-state /usr/lib/gcc/aarch64-linux-gnu/9/crtendS.o /usr/lib/gcc/aarch64-linux-
gnu/9/../../../aarch64-linux-gnu/crtn.o
```

main is not the actual entry point!

```
#0  f (x=1) at sin2cos2.c:5
#1  0x0000aaaaaaaa0848 in main (argc=1, argv=0xffffffff328) at sin2cos2.c:10
#2  0x0000ffff7dbae10 in __libc_start_main (main=0xaaaaaaaa082c <main>, argc=1, argv=0xffffffff328, init=<optimized out>,
    fini=<optimized out>, rtld_fini=<optimized out>, stack_end=<optimized out>) at ../csu/libc-start.c:308
#3  0x0000aaaaaaaa0704 in _start ()
```


Middle End
Front End
C
C++
Java
AST
AST
AST
Generic
GIMPLE
SSA
opt
pass 1
opt
pass N
un-SSA
RTL
machine
code
Back End

Why have 3 different IR’s

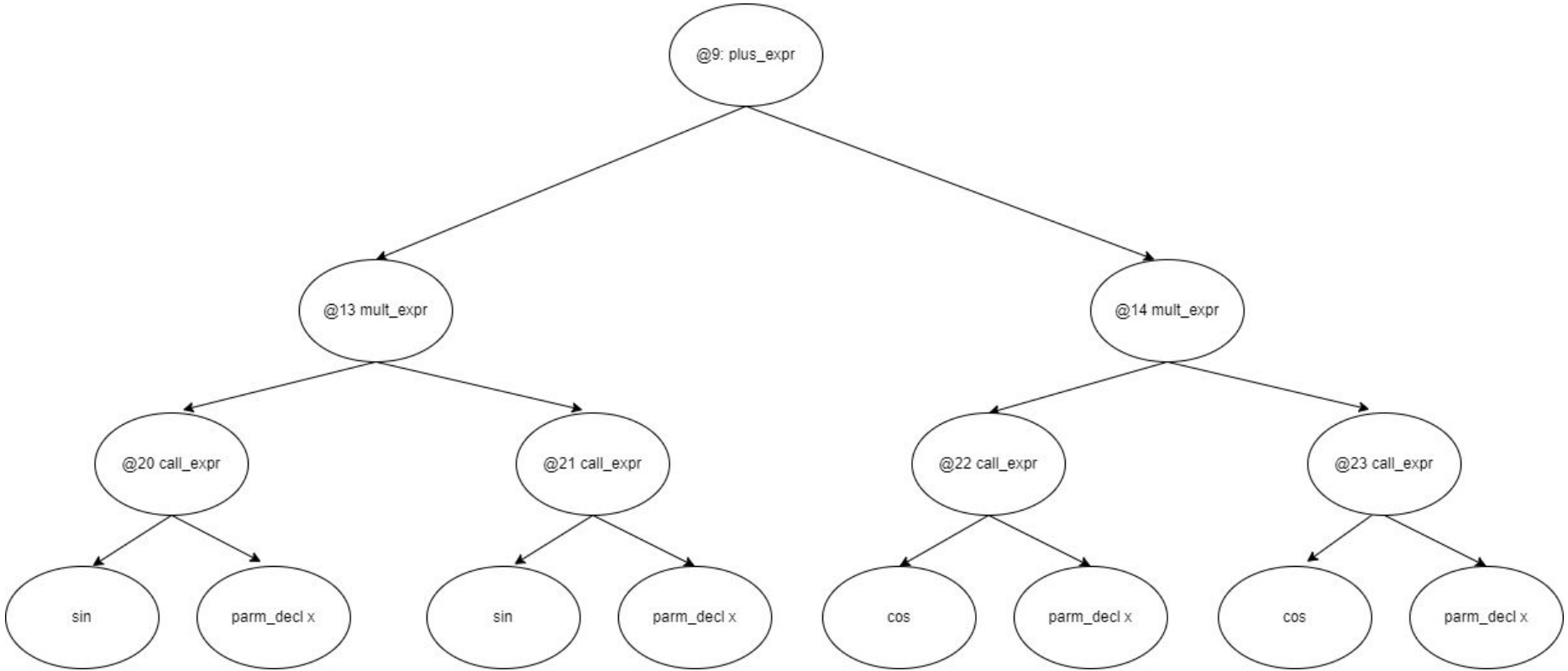
- GENERIC / AST
  - Rich representation of front-end constructs
  - Myopic, can only view one expression at a time
- GIMPLE IR
  - A “three address code” IR, akin to assembly
  - Has more context of program than GENERIC and thus better ability to optimize.
  - Sufficiently high level to abstract away target details
  - Most of target independent optimizations are performed at this level
- RTL
  - Ability to represent target specific constructs
  - Suitable for low level and target-specific optimizations

AST (GENERIC IR)

- The front-ends generate an AST from the input source after syntax analysis, and type checking
- AST provides a convenient way for other compiler components to access info gleaned from parsing.
- A tree is a natural structure because typical language constructs are nested.
- Let’s see AST generated by C front end for sin^2(x) + cos^2(x);

```
@9      plus_expr         type: @7       op 0: @13      op 1: @14
@10     type_decl         name: @15      type: @7
@11     integer_cst       type: @16      int: 64
@12     function_decl     name: @17      type: @18      srcp: sin2cos2.c:5
                          args: @19      link: extern
@13     mult_expr         type: @7       op 0: @20      op 1: @21
@14     mult_expr         type: @7       op 0: @22      op 1: @23
@15     identifier_node   strg: double   lngt: 6
@16     integer_type      name: @24      size: @25      algn: 128
                          prec: 128      sign: unsigned min : @26
                          max : @27
@17     identifier_node   strg: f        lngt: 1
@18     function_type     size: @28      algn: 32       retn: @7
                          prms: @29
@19     parm_decl         name: @30      type: @7       scpe: @12
                          srcp: sin2cos2.c:5            argt: @7
                          size: @11      algn: 64       used: 1
```

AST simplified view for sin^2(x) + cos^2(x)

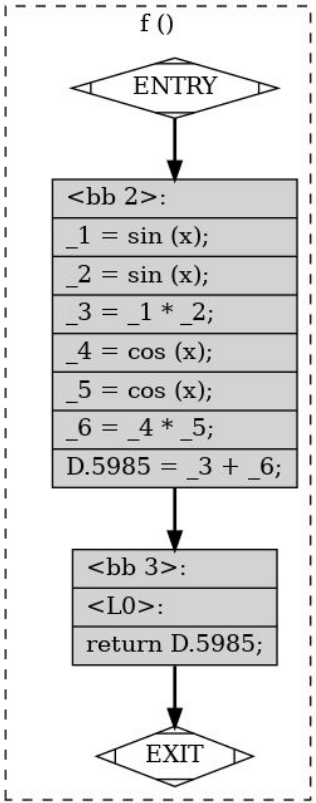


GIMPLE (Three Address Code)

```
double f (double x)
{
  double D.5985;

  _1 = sin (x);
  _2 = sin (x);
  _3 = _1 * _2;
  _4 = cos (x);
  _5 = cos (x);
  _6 = _4 * _5;
  D.5985 = _3 + _6;
  return D.5985;
}
```

- Each gimple statement is of “3 address code” form
  - At most two rhs operands and one lhs
  - GENERIC tree is lowered to multiple GIMPLE statements
- GIMPLE has more context of program than GENERIC

Control Flow Graph



- A basic block is a sequence of straight line code with no branches (except function calls)
- Node of CFG represents basic block
- Edges between basic blocks represent control flow
- Every CFG has two special nodes - ENTRY and EXIT
- Useful for identifying control flow within the program
    - Conditionals, loops etc.

RTL

- RTL is a low level IR that encodes target specific info.
- Used for low level program optimizations and code generation.
- List based.
- Divided into two kinds:
  - “Soft” RTL – Before register allocation in which, the IR allows for infinite “virtual” registers akin to GIMPLE’s temporary variables.
  - “Hard” RTL – After register allocation, when the virtual registers get mapped to physical registers of the machine.

RTL expansion

```
;; _6 = _4 * _5;

(insn 19 18 20 2 (set (reg:DF 105 [ _6 ])
        (mult:DF (reg:DF 103 [ _4 ])
            (reg:DF 104 [ _5 ]))) "sin2cos2.c":7:41 discrim 4 -1
     (nil))
```

Register Allocation

- Registers are like “very fast scratchpads” used by CPU
  - For storing result of computations
  - Loading and storing data from memory
  - A machine has limited number of registers
    - AArch64 for example defines 31 general purpose registers: x0 - x31
- Compiler performs code optimization assuming an “unlimited” number of registers.
  - Which typically creates large number of IR temporary variables.
- The Problem: Map IR temporaries (“virtual regs”) to physical registers.

RTL after register allocation

```
(insn 19 18 20 (set (reg:DF 63 v31 [orig:105 _6 ] [105])
        (mult:DF (reg:DF 46 v14 [orig:103 _4 ] [103])
            (reg:DF 63 v31 [orig:104 _5 ] [104]))) "sin2cos2.c":7:41 discrim 4 1050 {muldf3}
     (nil))
```

Code-gen: AArch64

```
f:
        stp     x29, x30, [sp, -48]!
        mov     x29, sp
        stp     d14, d15, [sp, 16]
        str     d0, [sp, 40]
        ldr     d0, [sp, 40]
        bl      sin
        fmov    d15, d0
        ldr     d0, [sp, 40]
        bl      sin
        fmov    d31, d0
        fmul    d15, d15, d31
        ldr     d0, [sp, 40]
        bl      cos
        fmov    d14, d0
        ldr     d0, [sp, 40]
        bl      cos
        fmov    d31, d0
        fmul    d31, d14, d31
        fadd    d31, d15, d31
        fmov    d0, d31
        ldp     d14, d15, [sp, 16]
        ldp     x29, x30, [sp], 48
        ret
```

- Calling convention: Pass floating point arguments in registers: d0-d7
- Four function calls
  - Two bl sin
  - Two bl cos
- Uses 2 fmul instructions to compute sin^2(x) and cos^2(x) respectively
- fadd will compute the final result for the identity.

Code-gen: x86_64

```
f:
        endbr64
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $32, %rsp
        movsd   %xmm0, -8(%rbp)
        movq    -8(%rbp), %rax
        movq    %rax, %xmm0
        call    sin@PLT
        movsd   %xmm0, -16(%rbp)
        movq    -8(%rbp), %rax
        movq    %rax, %xmm0
        call    sin@PLT
        movapd  %xmm0, %xmm1
        mulsd   -16(%rbp), %xmm1
        movsd   %xmm1, -16(%rbp)
        movq    -8(%rbp), %rax
        movq    %rax, %xmm0
        call    cos@PLT
        movsd   %xmm0, -24(%rbp)
        movq    -8(%rbp), %rax
        movq    %rax, %xmm0
        call    cos@PLT
        mulsd   -24(%rbp), %xmm0
        addsd   -16(%rbp), %xmm0
        movq    %xmm0, %rax
        movq    %rax, %xmm0
        leave
        ret
```

- GCC can generate code for multiple targets – AArch64, AArch32, x86_64 etc.

Can we do better than this ?

```
// Compute sin(x) * sin(x) into r0
r0 = sin(x);
r1 = sin(x);
r0 = r0 * r1;

// Compute cos(x) * cos(x) into r1
r1 = cos(x);
r2 = cos(x);
r1 = r1 * r2;

// Store final result into r0
r0 = r0 + r1;
```

Remove redundant calls to sin and cos

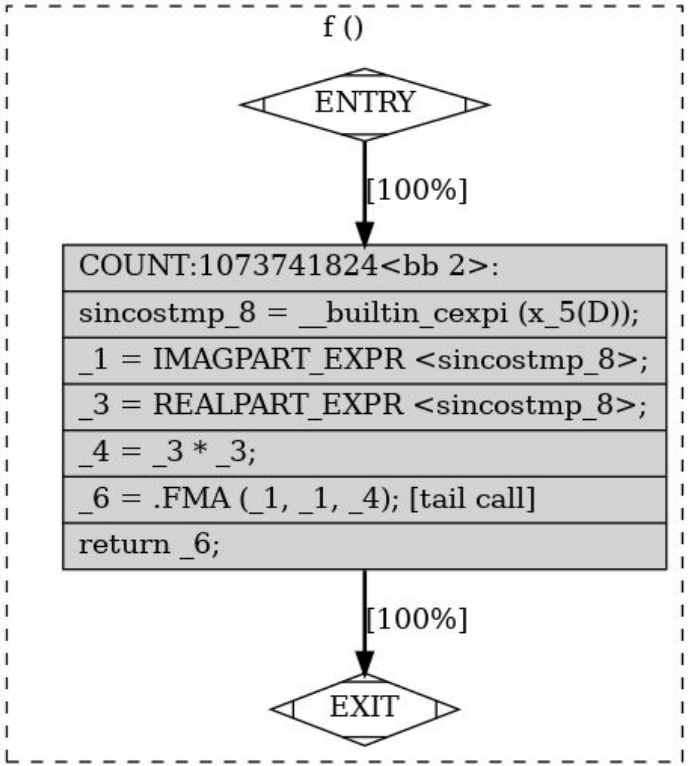
```
// Compute sin(x) * sin(x) into r0
r0 = sin(x);
r0 = r0 * r0;

// Compute cos(x) * cos(x) into r1
r1 = cos(x);
r1 = r1 * r1;

// Store final result into r0
r0 = r0 + r1;
```

Can we do still better ?

Can we do still better ?

- Euler’s identity!
  - e^x = sin(x) + i*cos(x)
- How can we use it to better optimize the code ?
  - Compute e^x
  - Extract real part of result to get sin(x) and multiply by itself to obtain sin^2(x)
  - Extract imaginary part of result to get cos(x) and multiply by itself to obtain cos^2(x)
  - Add sin^2(x) + cos^2(x)
- Compute sin and cos in one shot by computing e^x!

gcc -O2 does exactly this



- Took advantage of Euler's identity:
  - e^x = sin (x) + i*cos(x);
  - cexp(x) computes e^x
- Instead of computing sin(x) and cos (x) separately, emitted a single call to cexp(x) and extract real and imaginary parts for sin and cos
  - REALPART_EXPR<x> corresponds to real part of x
  - IMAGPART_EXPR<x> corresponds to imaginary part of x.
- .FMA is fused multiply and add
  - .FMA(x, y, z) = x*y + z
  - Computes sin(x) * sin(x) and adds it to result of cos(x) * cos(x)

Code-gen: -O0 vs -O2

```
f:
        stp     x29, x30, [sp, -48]!
        mov     x29, sp
        stp     d14, d15, [sp, 16]
        str     d0, [sp, 40]
        ldr     d0, [sp, 40]
        bl      sin
        fmov    d15, d0
        ldr     d0, [sp, 40]
        bl      sin
        fmov    d31, d0
        fmul    d15, d15, d31
        ldr     d0, [sp, 40]
        bl      cos
        fmov    d14, d0
        ldr     d0, [sp, 40]
        bl      cos
        fmov    d31, d0
        fmul    d31, d14, d31
        fadd    d31, d15, d31
        fmov    d0, d31
        ldp     d14, d15, [sp, 16]
        ldp     x29, x30, [sp], 48
        ret
```

```
f:
        stp     x29, x30, [sp, -32]!
        mov     x29, sp
        add     x1, sp, 16
        add     x0, sp, 24
        bl      sincos
        ldp     d31, d0, [sp, 16]
        ldp     x29, x30, [sp], 32
        fmul    d31, d31, d31
        fmadd   d0, d0, d0, d31
        ret
```

Peephole transforms

- A peephole optimization is one in which a sequence of straight-line code is replaced with “faster” sequence.
- Some examples
  - x + 0 → x
  - abs(abs(x)) → abs(x)
- We want to implement the following identity in compiler:
  - sin(x) * sin(x) + cos(x) * cos(x) → 1.0

Peephole transforms in GCC

- GCC provides a DSL match.pd for implementing peephole patterns for GENERIC and GIMPLE IR’s
  - This makes it easier to write the transforms in a special purpose language than directly manipulating IR’s with C++ API
  - Since GIMPLE and GENERIC share lot of infrastructure, we can write pattern once and have it generated for both IR’s. Targeting RTL at same time is much harder (and not done).
  - The DSL is written in match.pd and the corresponding generator program is genmatch.cc
- Our task is to write the sin^2(x) + cos^2(x) pattern using match.pd syntax

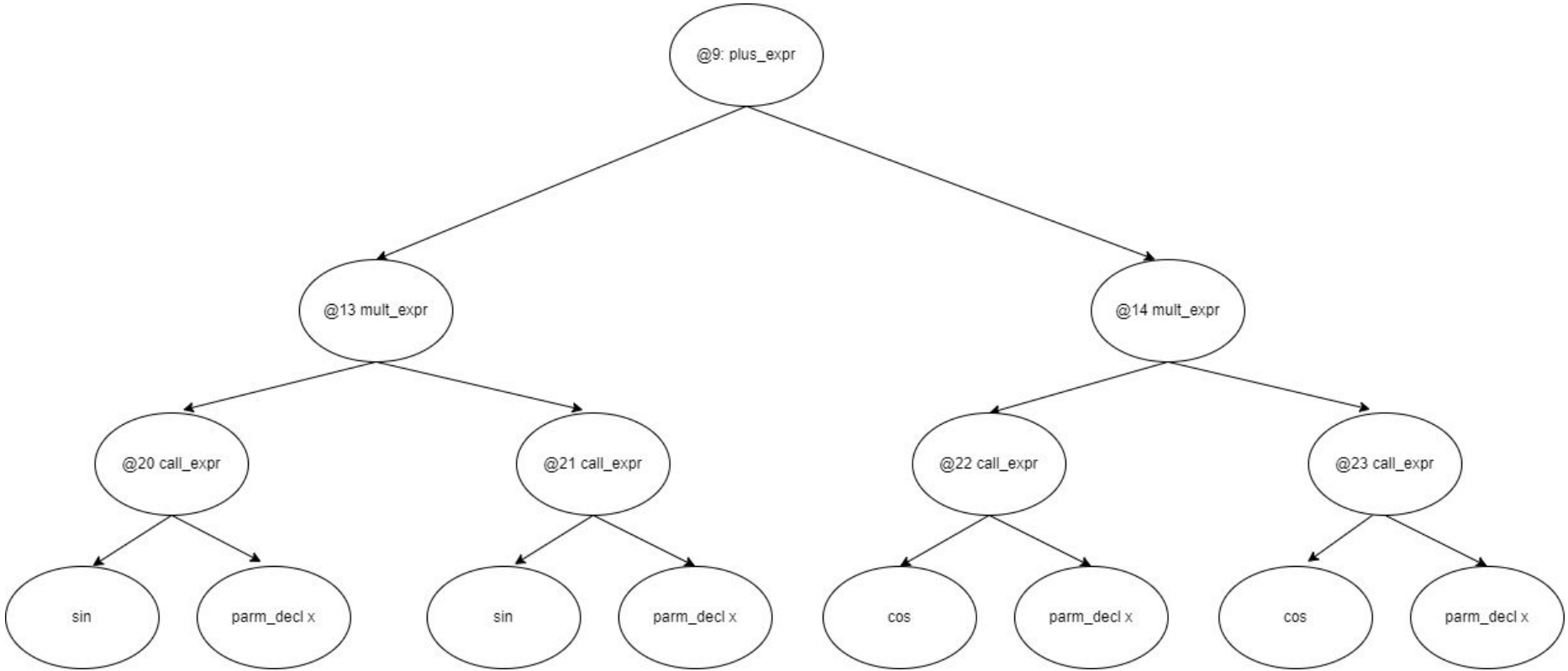
Pattern in match.pd syntax

```
(simplify
 (plus (mult (SIN @0) (SIN @0))
       (mult (COS @0) (COS @0)))
 (if (flag_unsafe_math_optimizations)
  { build_one_cst (TREE_TYPE (@0)); }))
```

Pattern in match.pd syntax

- Each pattern in match.pd starts with (simplify… or (match…
- The operators in pattern map to corresponding tree codes that uniquely identify trees.
- Written in a lisp-style format:
  - The first part is the so called “match”, which pattern matches a code sequence in either GENERIC or GIMPLE
  - The second part is the if condition predicate on which the pattern is gated – Even if the code sequence matches successfully, it has to additionally pass the predicate to satisfy any additional constraints.
  - The last part is the so called “transform”, which is C++ code to build replacement code-sequence

AST before transform



AST after transform

Code-gen

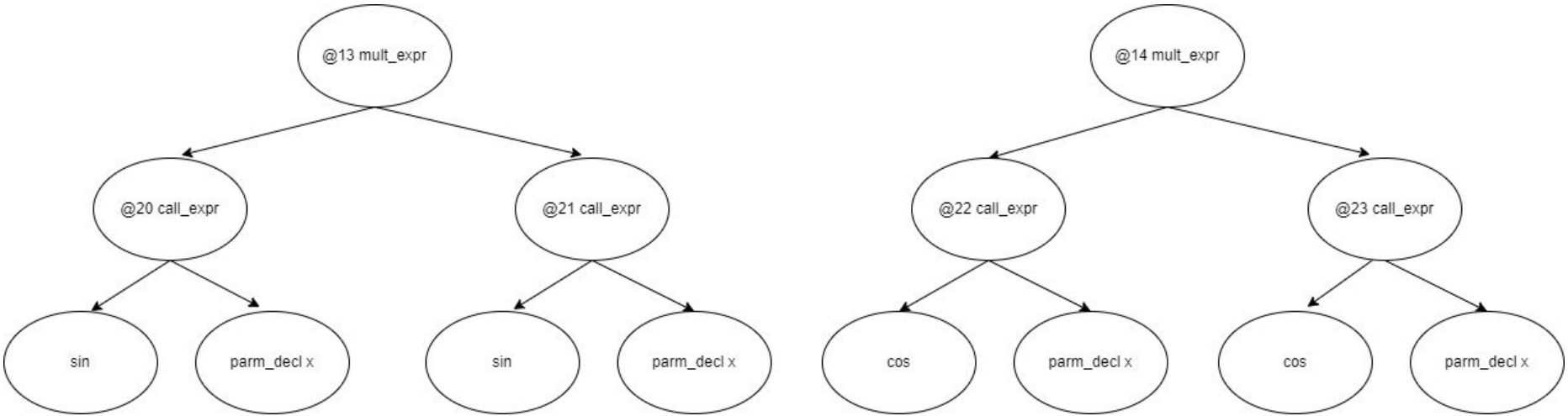
```
f:
        fmov    d0, 1.0e+0
        ret
```

Let’s see what happens when written this way

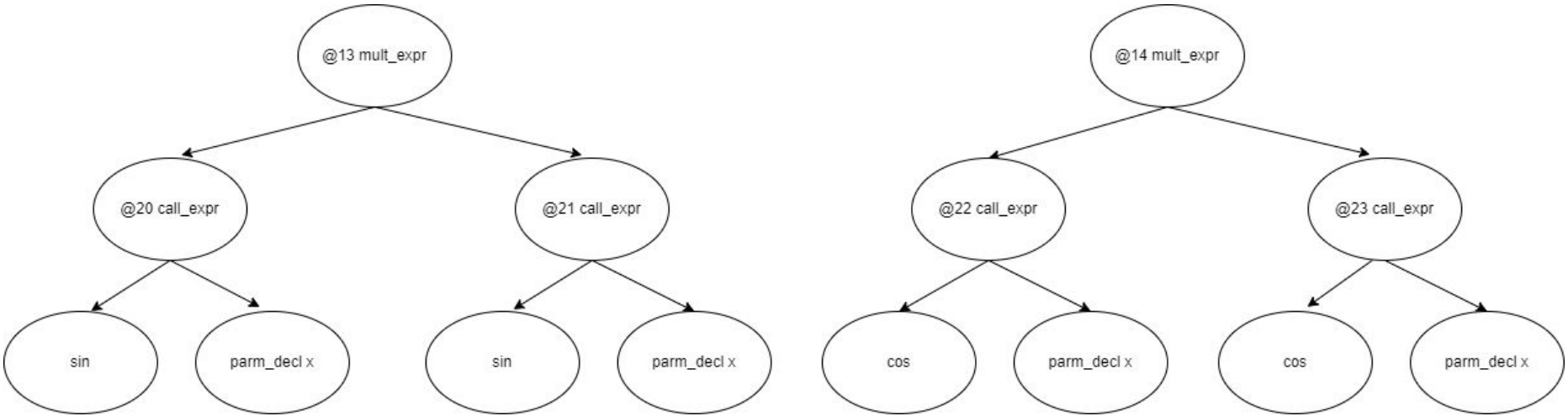
```c
#include <math.h>

double f(double x)
{
  double r1 = sin (x) * sin (x);
  double r2 = cos (x) * cos (x);
  return r1 + r2;
}
```

-

AST before transform



AST after transform



GIMPLE IR representation

```
double f (double x)
{
  double D.5522;
  double r1;
  double r2;

  _1 = sin (x);
  _2 = sin (x);
  r1 = _1 * _2;
  _3 = cos (x);
  _4 = cos (x);
  r2 = _3 * _4;
  D.5522 = r1 + r2;
  return D.5522;
}
```

Transform happens during ccp1 GIMPLE pass

```
Applying pattern match.pd:10469, gimple-match-7.cc:885
Match-and-simplified r1_6 + r2_7 to 1.0e+0

double f (double x)
{
  double r2;
  double r1;

  <bb 2> :
  return 1.0e+0;

}
```

- ccp is “conditional constant propagation” pass, which propagates constants over the control flow graph.
- Fold gimple statements across basic block by using patterns in match.pd.
  - In this case it folded the sequence of sin(x) * sin(x) + cos(x) * cos(x) → 1

Code-gen

```
f:
        fmov    d0, 1.0e+0
        ret
```

Comparing code-gen

- Comparing code-gen for O0, O2, and with our transform:
  - Code-gen with O0 was longest with 4 function calls
  - Code-gen with O2 was shorter with just a single call to sincos and using FMA to compute sin and add it to result of cos * cos
  - Code-gen with our transform was shortest by simply returning 1
- Intuitively,
  - Code compiled with our transform should be fastest
  - Code compiled with O0 should be slowest
- But how do we verify this and express it quantitatively ?
  - “Faster than”, “slower than” is not precise, we want to know exactly how fast or slow a program became with a particular transform

Microbenchmark!

```c
#include <stdio.h>
#include <math.h>
#include <time.h>

__attribute__((noipa))
double f(double x)
{
  return (sin (x) * sin (x)) + (cos (x) * cos (x));
}

int main()
{
  clock_t start = clock ();
  for (volatile int i = 0; i < 500000000; i++)
    f (i);
  clock_t end = clock ();
  double time_in_sec = (double) (end - start) / CLOCKS_PER_SEC;
  printf ("Time taken: %g\n", time_in_sec);
}
```

- Call f inside a loop that iterates for a large upper bound.
- Attributes like noipa and volatile avoid introducing “noise” by disabling unrelated optimizations
  - We would like to pin differences to the code-gen for function ‘f’ as far as possible
- clock() returns an approximation of processor time used by the function
  - Record clock() before starting the loop
  - Record clock() after finishing the loop
  - The time difference will be (mostly) attributed due to the differences in code-gen for f.

Microbenchmark results

Optimization	Speedup
No Optimization	Baseline
Removing redundant calls to sin and cos	2x
Euler’s Identity	3x
With transform	175x

- Baseline is execution time of microbenchmark at -O0
- Absolute results are not important, since what we really want to see is performance uplift relative to baseline with our transform

Key takeaways

- GCC is a multi language, multi platform compiler sharing a common optimizer.
- Three different IR’s – GENERIC, GIMPLE and RTL having different levels of abstraction
- For an optimization, correctness trumps all other considerations!
- Even if an optimization may improve a particular test-case, it may not be always useful to implement in the compiler
- Benchmark your optimization to provide evidence for its usefulness.