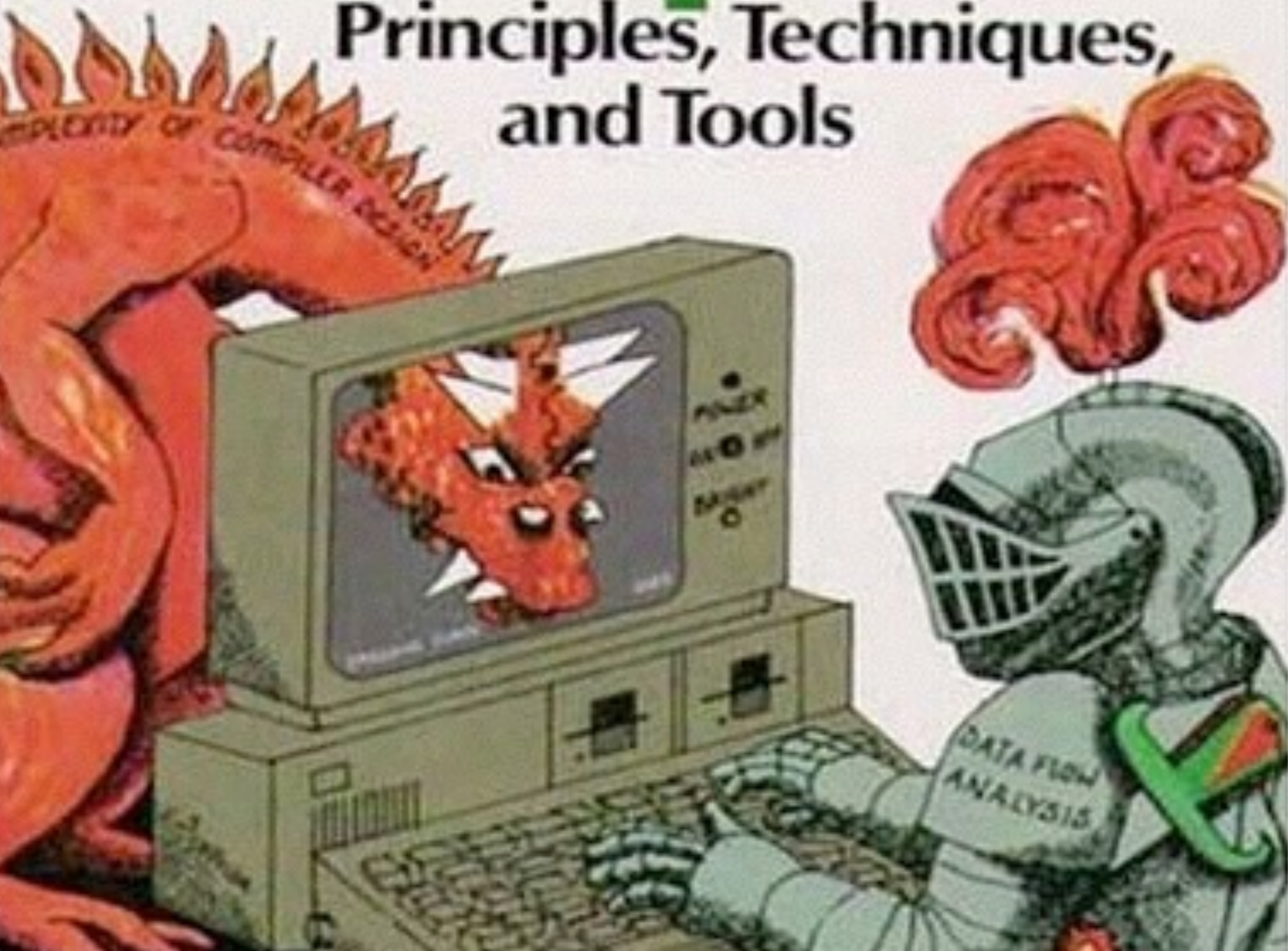
CHANDLER CARRUTH #MEETINGCPP 2015

UNDERSTANDING COMPILER OPTIMIZATION

Compilers

Principles, Techniques, and Tools



UNDERSTANDING PERFORMANCE MEANS UNDERSTANDING OPTIMIZERS

LET'S LOOK AT HOW COMPILERS ACTUALLY WORK…

FRONTEND

OPTIMIZER

CODE

OPTIMIZATION 101

```
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) + atoi(argv[2]);
}
```

```
TranslationUnitDecl 0x5332240 <<invalid sloc>>
|-LinkageSpecDecl 0x5332bf0 <hello_world.cpp:1:1, col:33> C
| `-FunctionDecl 0x5332d40 <col:12, col:33> atoi 'int (const char *)'
|   `-ParmVarDecl 0x5332c80 <col:21, col:33> 'const char *'
`-FunctionDecl 0x535e680 <line:2:1, line:7:1> main 'int (int, char **)'
  |-ParmVarDecl 0x5332e00 <line:2:10, col:14> argc 'int'
  |-ParmVarDecl 0x5332ed0 <col:20, col:27> argv 'char **'
  `-CompoundStmt 0x535ebb8 <col:33, line:7:1>
    |-IfStmt 0x535e818 <line:3:3, line:4:13>
    | |-<<<NULL>>>
    | |-BinaryOperator 0x535e790 <line:3:7, col:15> '_Bool' '!='
    | | |-ImplicitCastExpr 0x535e778 <col:7> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0x535e730 <col:7> 'int' lvalue ParmVar 0x5332e00 'argc' 'int'
    | | `-IntegerLiteral 0x535e758 <col:15> 'int' 3
    | |-ReturnStmt 0x535e7f8 <line:4:5, col:13>
    | | `-UnaryOperator 0x535e7d8 <col:12, col:13> 'int' prefix '-'
    | |   `-IntegerLiteral 0x535e7b8 <col:13> 'int' 1
    | `-<<<NULL>>>
    `-ReturnStmt 0x535eb98 <line:6:3, col:38>
      `-BinaryOperator 0x535eb70 <col:10, col:38> 'int' '+'
        |-CallExpr 0x535e990 <col:10, col:22> 'int'
        | |-ImplicitCastExpr 0x535e978 <col:10> 'int (*)(const char *)' <FunctionToPointerDecay>
        | | `-DeclRefExpr 0x535e928 <col:10> 'int (const char *)' lvalue Function 0x5332d40 'atoi' '...'
        | `-ImplicitCastExpr 0x535e9d8 <col:15, col:21> 'const char *' <NoOp>
        |   `-ImplicitCastExpr 0x535e9c0 <col:15, col:21> 'char *' <LValueToRValue>
        |     `-ArraySubscriptExpr 0x535e900 <col:15, col:21> 'char *' lvalue
        |       |-ImplicitCastExpr 0x535e8e8 <col:15> 'char **' <LValueToRValue>
        |       | `-DeclRefExpr 0x535e8a0 <col:15> 'char **' lvalue ParmVar 0x5332ed0 'argv' 'char **'
        |       `-IntegerLiteral 0x535e8c8 <col:20> 'int' 1
        `-CallExpr 0x535eb10 <col:26, col:38> 'int'
          |-ImplicitCastExpr 0x535eaf8 <col:26> 'int (*)(const char *)' <FunctionToPointerDecay>
          | `-DeclRefExpr 0x535ead0 <col:26> 'int (const char *)' lvalue Function 0x5332d40 'atoi' 'int (const char
*)'
```

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  %0 = load i32* %argc.addr, align 4
  %cmp = icmp ne i32 %0, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  store i32 -1, i32* %retval
  br label %return

if.end:                                           ; preds = %entry
  %1 = load i8*** %argv.addr, align 8
  %arrayidx = getelementptr inbounds i8** %1, i64 1
  %2 = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %2)
  %3 = load i8*** %argv.addr, align 8
  %arrayidx1 = getelementptr inbounds i8** %3, i64 2
  %4 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %4)
  %add = add nsw i32 %call, %call2
  store i32 %add, i32* %retval
  br label %return

return:                                           ; preds = %if.end, %if.then
  %5 = load i32* %retval
  ret i32 %5
}
```

A BRIEF DIGRESSION TO DESCRIBE LLVM’S IR…

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b) {
entry:
  %c = add i32 %a, %b
  %d = call i32 @g(i32 %c)
  %e = add i32 %c, %d
  ret i32 %e
}
```

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b, i1 %flag) {
entry:
  %c = add i32 %a, %b
  br i1 %flag, label %then, label %else

then:
  %d = call i32 @g(i32 %c)
  ret i32 %d

else:
  ret i32 %c
}
```

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b, i1 %flag) {
entry:
  %c = add i32 %a, %b
  br i1 %flag, label %then, label %end

then:
  %d = call i32 @g(i32 %c)
  br label %end

end:
  %result = phi i32 [ %entry, %c ],
                    [ %then, %d ]
  ret i32 %result
}
```

OK, WHERE WERE WE…
IR FOR HELLO WORLD

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  %0 = load i32* %argc.addr, align 4
  %cmp = icmp ne i32 %0, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  store i32 -1, i32* %retval
  br label %return

if.end:                                           ; preds = %entry
  %1 = load i8*** %argv.addr, align 8
  %arrayidx = getelementptr inbounds i8** %1, i64 1
  %2 = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %2)
  %3 = load i8*** %argv.addr, align 8
  %arrayidx1 = getelementptr inbounds i8** %3, i64 2
  %4 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %4)
  %add = add nsw i32 %call, %call2
  store i32 %add, i32* %retval
  br label %return

return:                                           ; preds = %if.end, %if.then
  %5 = load i32* %retval
  ret i32 %5
}
```

OPTIMIZATION DOES MORE THAN JUST MAKE CODE FASTER...

STEP 1: CLEANUP

```c
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) +
         atoi(argv[2]);
}
```

```llvm
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  %0 = load i32* %argc.addr, align 4
  %cmp = icmp ne i32 %0, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:
  store i32 -1, i32* %retval
  br label %return

if.end:
  %1 = load i8*** %argv.addr, align 8
  %arrayidx = getelementptr i8** %1, i64 1
  %2 = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %2)
  %3 = load i8*** %argv.addr, align 8
  %arrayidx1 = getelementptr i8** %3, i64 2
  %4 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %4)
  %add = add nsw i32 %call, %call2
  store i32 %add, i32* %retval
  br label %return

return:
  %5 = load i32* %retval
  ret i32 %5
}
```

```c
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) +
         atoi(argv[2]);
}
```

```llvm
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %cmp = icmp ne i32 %argc, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:
  br label %return

if.end:
  %arrayidx = getelementptr i8** %argv, i64 1
  %arrayval = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %arrayval)
  %arrayidx1 = getelementptr i8** %argv, i64 2
  %arrayval1 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %arrayval1)
  %add = add nsw i32 %call, %call2
  br label %return

return:
  %retval.0 = phi i32 [ -1, %if.then ],
                      [ %add, %if.end ]
  ret i32 %retval.0
}
```

STEP 2: CANONICALIZATION

```
int x = y;
if (!flag)
  x = z;
```

```
int x;
if (flag)
  x = y;
else
  x = z;
```

```
if (flag)
  z = y;
int x = z;
```

```
int x = flag? y : z;
```

```c
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) +
         atoi(argv[2]);
}
```

```llvm
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %cmp = icmp ne i32 %argc, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:
  br label %return

if.end:
  %arrayidx = getelementptr i8** %argv, i64 1
  %arrayval = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %arrayval)
  %arrayidx1 = getelementptr i8** %argv, i64 2
  %arrayval1 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %arrayval1)
  %add = add nsw i32 %call, %call2
  br label %return

return:
  %retval.0 = phi i32 [ -1, %if.then ],
                      [ %add, %if.end ]
  ret i32 %retval.0
}
```

```cpp
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) +
         atoi(argv[2]);
}
```

```llvm
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %cmp = icmp eq i32 %argc, 3
  br i1 %cmp, label %if.end, label %if.then

if.then:
  br label %return

if.end:
  %arrayidx = getelementptr i8** %argv, i64 1
  %arrayval = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %arrayval)
  %arrayidx1 = getelementptr i8** %argv, i64 2
  %arrayval1 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %arrayval1)
  %add = add nsw i32 %call, %call2
  br label %return

return:
  %retval.0 = phi i32 [ -1, %if.then ],
                      [ %add, %if.end ]
  ret i32 %retval.0
}
```

```c
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) +
         atoi(argv[2]);
}
```

```llvm
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %cmp = icmp eq i32 %argc, 3
  br i1 %cmp, label %if.end, label %return

if.end:
  %arrayidx = getelementptr i8** %argv, i64 1
  %arrayval = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %arrayval)
  %arrayidx1 = getelementptr i8** %argv, i64 2
  %arrayval1 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %arrayval1)
  %add = add nsw i32 %call, %call2
  br label %return

return:
  %retval.0 = phi i32 [ -1, %entry ],
                      [ %add, %if.end ]
  ret i32 %retval.0
}
```

STEP 3: COLLAPSE ABSTRACTIONS

THREE KEY ABSTRACTIONS:

1. Functions, calls, and the call graph.
2. Memory, loads, and stores.
3. Loops.

WHAT ABOUT THE OTHER FUNDAMENTAL OPTIMIZATIONS?

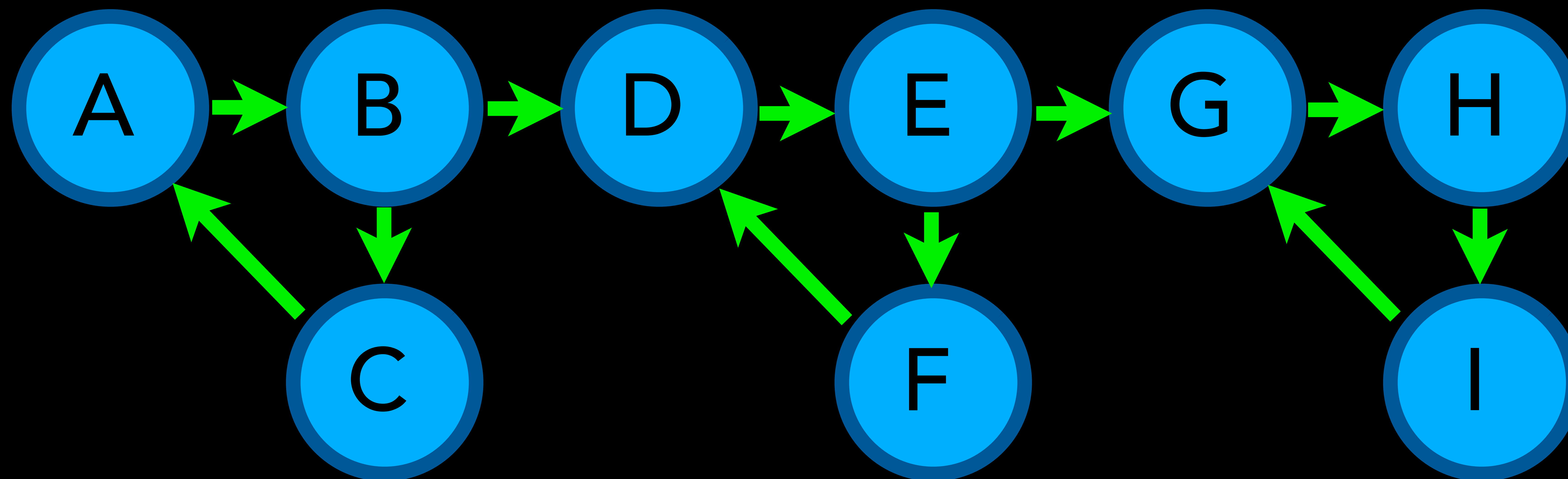
LET’S LOOK AT FUNCTION CALLS

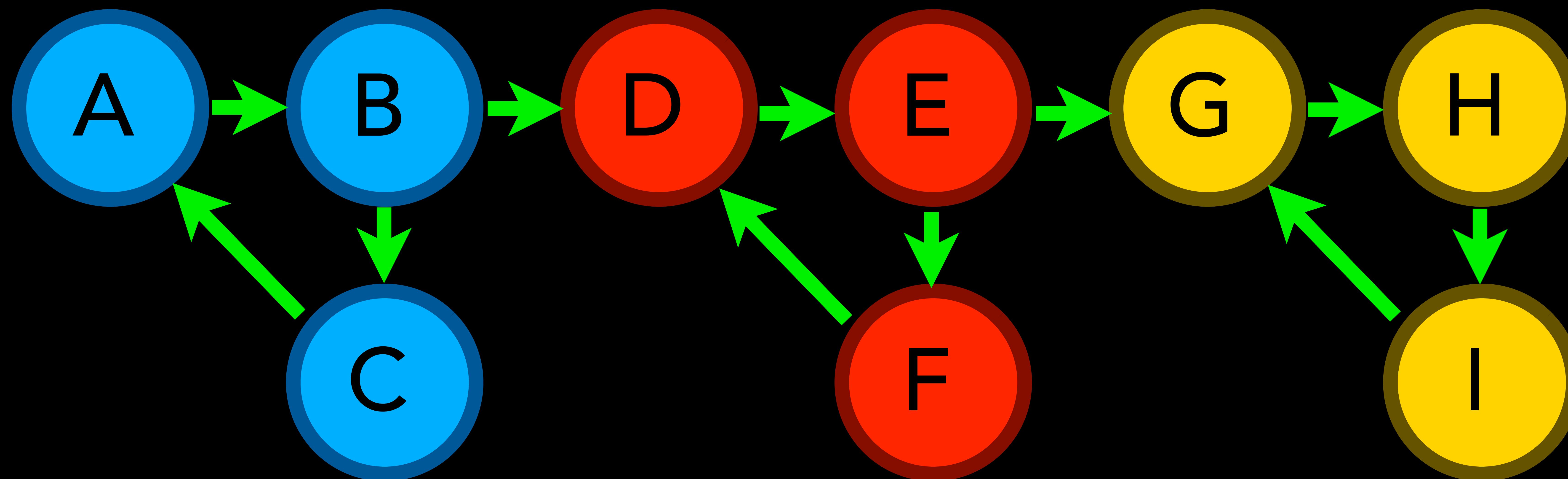
THE OPTIMIZATION TO COLLAPSE A FUNCTION CALL IS CALLED "INLINING"

THIS IS THE SINGLE MOST IMPORTANT OPTIMIZATION IN MODERN COMPILERS

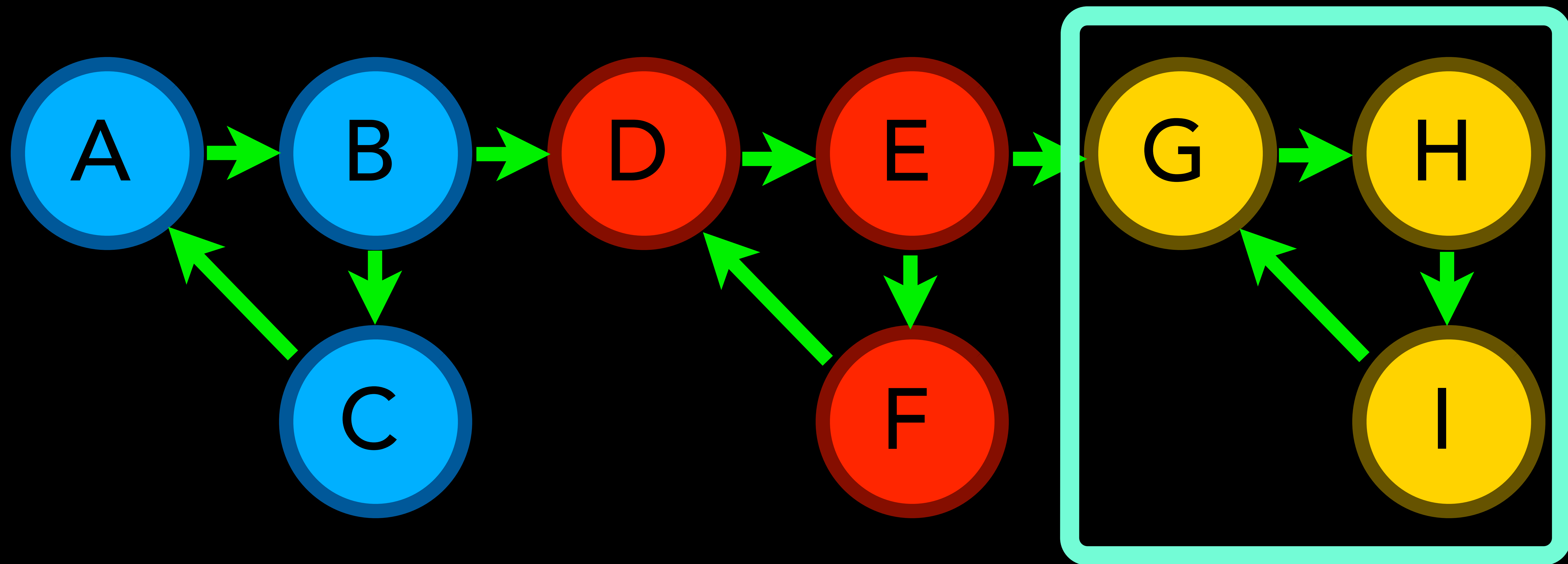
INLINING HAS THE GOLDILOCKS PROBLEM…

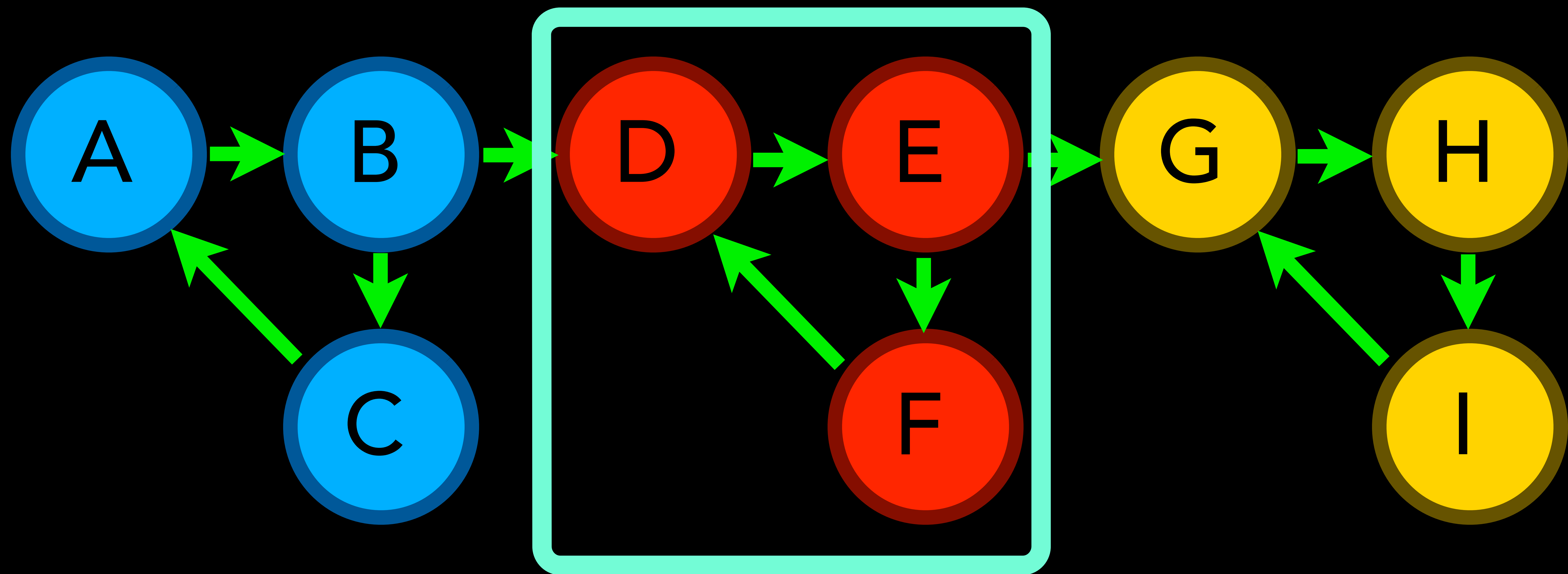
INLINING TOO MUCH...

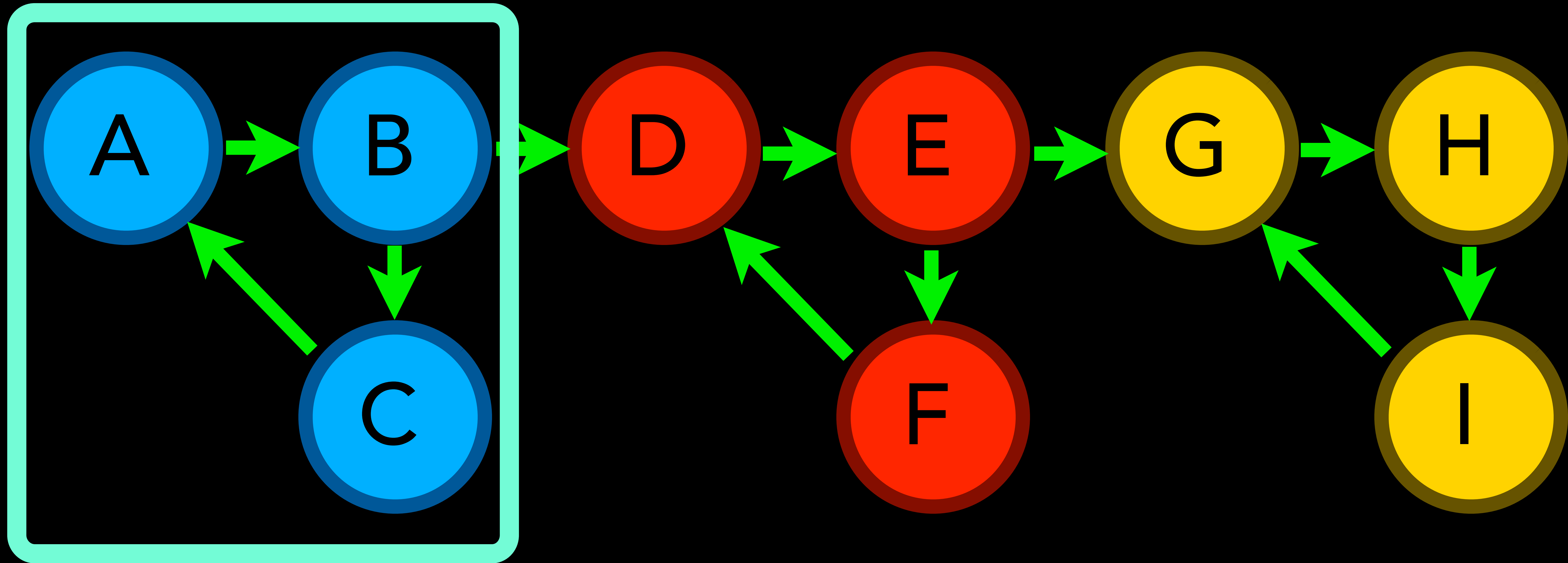

A
B
D
E
G
H
C
F
I


A
B
D
E
G
H
C
F
I

BOTTOM-UP SCC-BASED CALL GRAPH WALK


A
B
D
E
G
H
C
F
I


A
B
C
D
E
F
G
H
I


A
B
C
D
E
F
G
H
I

HOW DOES THE OPTIMIZER EVALUATE COMPLEXITY?

```c
int g(double x, double y, double z);

int f(struct S* s, double y, double x) {
  return g(x, y, s->z);
}
```

```
void fancy_sort(vector<int> &v) {
  if (v.size() <= 1)
    return;
  if (v.size() == 2) {
    if (v.front() >= v.back())
      swap(v.front(), v.back());
    return;
  }
  std::sort(v.begin(), v.end());
}
```

THIS DOESN’T ALWAYS WORK THOUGH!

```cpp
int hash(hash_state &h) {
  // Some complex code on 'h'
  return /* final value */;
}

template <typename T, typename ...Ts>
int hash(hash_state &h, T arg, Ts ...args) {
  // Complex code to put 'arg'
  // into the 'h' state...
  return hash(h, args...);
}
```

LET’S LOOK AT MEMORY, LOADS, AND STORES…

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b,
              i1 %flag) {
entry:
  %mem = alloca i32
  %c = add i32 %a, %b
  store i32 %c, i32* %mem
  br i1 %flag, label %then,
               label %end

then:
  %d = call i32 @g(i32 %c)
  store i32 %d, i32* %mem
  br label %end

end:
  %result = load i32* %mem
  ret i32 %result
}
```

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b,
              i1 %flag) {
entry:
  %c = add i32 %a, %b
  br i1 %flag, label %then,
               label %end

then:
  %d = call i32 @g(i32 %c)
  br label %end

end:
  %result = phi i32 [ %entry, %c ],
                    [ %then, %d ]
  ret i32 %result
}
```

```
%S = type { i32, i32, i32 }
declare i32 @g(i32 %x)
define i32 @f(i32 %a, i32 %b, i1 %flag) {
entry:
  %mem = alloca %S
  %c = add i32 %a, %b
  %addr0 = getelementptr %S* %mem,
                         i32 0, i32 0
  store i32 %c, i32* %addr0
  %addr1 = getelementptr %S* %mem,
                         i32 0, i32 1
  store i32 0, i32* %addr1
  %addr2 = getelementptr %S* %mem,
                         i32 0, i32 2
  store i32 0, i32* %addr2
  br i1 %flag, label %then, label %end

then:
  %d = call i32 @g(i32 %c)
  store i32 %d, i32* %addr1
  %e = call i32 @g(i32 %a)
  store i32 %e, i32* %addr2
  br label %end

end:
  %val0 = load i32* %addr0
  %val1 = load i32* %addr1
  %val2 = load i32* %addr2
  %f = add i32 %val0, %val1
  %result = add i32 %f, %val2
  ret i32 %result
}
```

```
declare i32 @g(i32)

define i32 @f(i32 %a, i32 %b, i1 %flag) {
entry:
  %c = add i32 %a, %b
  br i1 %flag, label %then, label %end

then:
  %d = call i32 @g(i32 %c)
  %e = call i32 @g(i32 %a)
  br label %end

end:
  %mem.sroa.1.0 =
      phi i32 [ %d, %then ], [ 0, %entry ]
  %mem.sroa.2.0 =
      phi i32 [ %e, %then ], [ 0, %entry ]
  %f = add i32 %c, %mem.sroa.1.0
  %result = add i32 %f, %mem.sroa.2.0
  ret i32 %result
}
```

DO WE REALLY NEED TO LOOK AT LOOPS?

Fortran

BECAUSE
IF YOU REALLY CARE…

C++ IS INCREASINGLY USED FOR MATH: HTTP://EIGEN.TUXFAMILY.ORG/

```
int sum(std::vector<int> &v) {
  int x = 0;
  for (auto i : v)
    x += i;
  return x;
}
```

```llvm
; ModuleID = 'loop_base.cpp'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

%"class.std::__1::vector" = type { %"class.std::__1::__vector_base" }
%"class.std::__1::__vector_base" = type { i32*, i32*, %"class.std::__1::__compressed_pair" }
%"class.std::__1::__compressed_pair" = type { %"class.std::__1::__libcpp_compressed_pair_imp" }
%"class.std::__1::__libcpp_compressed_pair_imp" = type { i32* }

; Function Attrs: norecurse nounwind readonly uwtable
define i32 @_Z3sumRNSt3__16vectorIiNS_9allocatorIiEEEE(%"class.std::__1::vector"* nocapture readonly dereferenceable(24) %v) #0 {
  %1 = getelementptr inbounds %"class.std::__1::vector", %"class.std::__1::vector"* %v, i64 0, i32 0, i32 0
  %2 = load i32*, i32** %1, align 8, !tbaa !1
  %3 = getelementptr inbounds %"class.std::__1::vector", %"class.std::__1::vector"* %v, i64 0, i32 0, i32 1
  %4 = load i32*, i32** %3, align 8, !tbaa !7
  %5 = icmp eq i32* %2, %4
  br i1 %5, label %._crit_edge, label %.lr.ph.preheader

.lr.ph.preheader:                                 ; preds = %0
  %6 = ptrtoint i32* %2 to i64
  %scevgep = getelementptr i32, i32* %4, i64 -1
  %7 = ptrtoint i32* %scevgep to i64
  %8 = sub i64 %7, %6
  %9 = lshr i64 %8, 2
  %10 = add nuw nsw i64 %9, 1
  %min.iters.check = icmp ult i64 %10, 16
  br i1 %min.iters.check, label %.lr.ph.preheader46, label %min.iters.checked

.lr.ph.preheader46:                               ; preds = %middle.block, %min.iters.checked, %.lr.ph.preheader
  %x.02.ph = phi i32 [ 0, %min.iters.checked ], [ 0, %.lr.ph.preheader ], [ %49, %middle.block ]
  %__begin.sroa.0.01.ph = phi i32* [ %2, %min.iters.checked ], [ %2, %.lr.ph.preheader ], [ %ind.end, %middle.block ]
  br label %.lr.ph

min.iters.checked:                                ; preds = %.lr.ph.preheader
  %n.vec = and i64 %10, 9223372036854775792
  %cmp.zero = icmp eq i64 %n.vec, 0
  %ind.end = getelementptr i32, i32* %2, i64 %n.vec
  br i1 %cmp.zero, label %.lr.ph.preheader46, label %vector.body.preheader

vector.body.preheader:                            ; preds = %min.iters.checked
  %11 = sub i64 %7, %6
  %12 = lshr i64 %11, 2
  %13 = add nuw nsw i64 %12, 1
  %14 = and i64 %13, 9223372036854775792
  %15 = add nsw i64 %14, -16
  %16 = lshr exact i64 %15, 4
  %17 = and i64 %16, 1
  %lcmp.mod = icmp eq i64 %17, 0
  br i1 %lcmp.mod, label %vector.body.prol, label %vector.body.preheader.split

vector.body.prol:                                 ; preds = %vector.body.preheader
  %18 = bitcast i32* %2 to <4 x i32>*
  %wide.load.prol = load <4 x i32>, <4 x i32>* %18, align 4, !tbaa !8
  %19 = getelementptr i32, i32* %2, i64 4
  %20 = bitcast i32* %19 to <4 x i32>*
  %wide.load38.prol = load <4 x i32>, <4 x i32>* %20, align 4, !tbaa !8
  %21 = getelementptr i32, i32* %2, i64 8
  %22 = bitcast i32* %21 to <4 x i32>*
  %wide.load39.prol = load <4 x i32>, <4 x i32>* %22, align 4, !tbaa !8
  %23 = getelementptr i32, i32* %2, i64 12
  %24 = bitcast i32* %23 to <4 x i32>*
  %wide.load40.prol = load <4 x i32>, <4 x i32>* %24, align 4, !tbaa !8
  br label %vector.body.preheader.split

vector.body.preheader.split:                      ; preds = %vector.body.prol, %vector.body.preheader
  %.lcssa50.unr = phi <4 x i32> [ undef, %vector.body.preheader ], [ %wide.load40.prol, %vector.body.prol ]
  %.lcssa49.unr = phi <4 x i32> [ undef, %vector.body.preheader ], [ %wide.load39.prol, %vector.body.prol ]
  %.lcssa48.unr = phi <4 x i32> [ undef, %vector.body.preheader ], [ %wide.load38.prol, %vector.body.prol ]
  %.lcssa47.unr = phi <4 x i32> [ undef, %vector.body.preheader ], [ %wide.load.prol, %vector.body.prol ]
  %index.unr = phi i64 [ 0, %vector.body.preheader ], [ 16, %vector.body.prol ]
  %vec.phi.unr = phi <4 x i32> [ zeroinitializer, %vector.body.preheader ], [ %wide.load.prol, %vector.body.prol ]
  %vec.phi5.unr = phi <4 x i32> [ zeroinitializer, %vector.body.preheader ], [ %wide.load38.prol, %vector.body.prol ]
  %vec.phi6.unr = phi <4 x i32> [ zeroinitializer, %vector.body.preheader ], [ %wide.load39.prol, %vector.body.prol ]
  %vec.phi7.unr = phi <4 x i32> [ zeroinitializer, %vector.body.preheader ], [ %wide.load40.prol, %vector.body.prol ]
  %25 = icmp eq i64 %16, 0
  br i1 %25, label %middle.block, label %vector.body.preheader.split.split

vector.body.preheader.split.split:                ; preds = %vector.body.preheader.split
  br label %vector.body

vector.body:                                      ; preds = %vector.body, %vector.body.preheader.split.split
  %index = phi i64 [ %index.unr, %vector.body.preheader.split.split ], [ %index.next.1, %vector.body ]
  %vec.phi = phi <4 x i32> [ %vec.phi.unr, %vector.body.preheader.split.split ], [ %44, %vector.body ]
  %vec.phi5 = phi <4 x i32> [ %vec.phi5.unr, %vector.body.preheader.split.split ], [ %45, %vector.body ]
  %vec.phi6 = phi <4 x i32> [ %vec.phi6.unr, %vector.body.preheader.split.split ], [ %46, %vector.body ]
  %vec.phi7 = phi <4 x i32> [ %vec.phi7.unr, %vector.body.preheader.split.split ], [ %47, %vector.body ]
  %next.gep = getelementptr i32, i32* %2, i64 %index
  %26 = bitcast i32* %next.gep to <4 x i32>*
  %wide.load = load <4 x i32>, <4 x i32>* %26, align 4, !tbaa !8
  %27 = getelementptr i32, i32* %next.gep, i64 4
  %28 = bitcast i32* %27 to <4 x i32>*
  %wide.load38 = load <4 x i32>, <4 x i32>* %28, align 4, !tbaa !8
  %29 = getelementptr i32, i32* %next.gep, i64 8
  %30 = bitcast i32* %29 to <4 x i32>*
  %wide.load39 = load <4 x i32>, <4 x i32>* %30, align 4, !tbaa !8
  %31 = getelementptr i32, i32* %next.gep, i64 12
  %32 = bitcast i32* %31 to <4 x i32>*
  %wide.load40 = load <4 x i32>, <4 x i32>* %32, align 4, !tbaa !8
  %33 = add nsw <4 x i32> %wide.load, %vec.phi
  %34 = add nsw <4 x i32> %wide.load38, %vec.phi5
  %35 = add nsw <4 x i32> %wide.load39, %vec.phi6
  %36 = add nsw <4 x i32> %wide.load40, %vec.phi7
  %index.next = add i64 %index, 16
  %next.gep.1 = getelementptr i32, i32* %2, i64 %index.next
  %37 = bitcast i32* %next.gep.1 to <4 x i32>*
  %wide.load.1 = load <4 x i32>, <4 x i32>* %37, align 4, !tbaa !8
  %38 = getelementptr i32, i32* %next.gep.1, i64 4
  %39 = bitcast i32* %38 to <4 x i32>*
  %wide.load38.1 = load <4 x i32>, <4 x i32>* %39, align 4, !tbaa !8
  %40 = getelementptr i32, i32* %next.gep.1, i64 8
  %41 = bitcast i32* %40 to <4 x i32>*
  %wide.load39.1 = load <4 x i32>, <4 x i32>* %41, align 4, !tbaa !8
  %42 = getelementptr i32, i32* %next.gep.1, i64 12
  %43 = bitcast i32* %42 to <4 x i32>*
  %wide.load40.1 = load <4 x i32>, <4 x i32>* %43, align 4, !tbaa !8
  %44 = add nsw <4 x i32> %wide.load.1, %33
  %45 = add nsw <4 x i32> %wide.load38.1, %34
  %46 = add nsw <4 x i32> %wide.load39.1, %35
  %47 = add nsw <4 x i32> %wide.load40.1, %36
  %index.next.1 = add i64 %index, 32
  %48 = icmp eq i64 %index.next.1, %n.vec
  br i1 %48, label %middle.block.unr-lcssa, label %vector.body, !llvm.loop !10

middle.block.unr-lcssa:                           ; preds = %vector.body
  %.lcssa54 = phi <4 x i32> [ %47, %vector.body ]
  %.lcssa53 = phi <4 x i32> [ %46, %vector.body ]
  %.lcssa52 = phi <4 x i32> [ %45, %vector.body ]
  %.lcssa51 = phi <4 x i32> [ %44, %vector.body ]
  br label %middle.block

middle.block:                                     ; preds = %vector.body.preheader.split, %middle.block.unr-lcssa
  %.lcssa50 = phi <4 x i32> [ %.lcssa50.unr, %vector.body.preheader.split ], [ %.lcssa54, %middle.block.unr-lcssa ]
  %.lcssa49 = phi <4 x i32> [ %.lcssa49.unr, %vector.body.preheader.split ], [ %.lcssa53, %middle.block.unr-lcssa ]
  %.lcssa48 = phi <4 x i32> [ %.lcssa48.unr, %vector.body.preheader.split ], [ %.lcssa52, %middle.block.unr-lcssa ]
  %.lcssa47 = phi <4 x i32> [ %.lcssa47.unr, %vector.body.preheader.split ], [ %.lcssa51, %middle.block.unr-lcssa ]
  %bin.rdx = add <4 x i32> %.lcssa48, %.lcssa47
  %bin.rdx41 = add <4 x i32> %.lcssa49, %bin.rdx
  %bin.rdx42 = add <4 x i32> %.lcssa50, %bin.rdx41
  %rdx.shuf = shufflevector <4 x i32> %bin.rdx42, <4 x i32> undef, <4 x i32> <i32 2, i32 3, i32 undef, i32 undef>
  %bin.rdx43 = add <4 x i32> %bin.rdx42, %rdx.shuf
  %rdx.shuf44 = shufflevector <4 x i32> %bin.rdx43, <4 x i32> undef, <4 x i32> <i32 1, i32 undef, i32 undef, i32 undef>
  %bin.rdx45 = add <4 x i32> %bin.rdx43, %rdx.shuf44
  %49 = extractelement <4 x i32> %bin.rdx45, i32 0
  %cmp.n = icmp eq i64 %10, %n.vec
  br i1 %cmp.n, label %._crit_edge, label %.lr.ph.preheader46

._crit_edge.loopexit:                             ; preds = %.lr.ph
  %.lcssa = phi i32 [ %51, %.lr.ph ]
  br label %._crit_edge

._crit_edge:                                      ; preds = %._crit_edge.loopexit, %middle.block, %0
  %x.0.lcssa = phi i32 [ 0, %0 ], [ %49, %middle.block ], [ %.lcssa, %._crit_edge.loopexit ]
  ret i32 %x.0.lcssa

.lr.ph:                                           ; preds = %.lr.ph.preheader46, %.lr.ph
  %x.02 = phi i32 [ %51, %.lr.ph ], [ %x.02.ph, %.lr.ph.preheader46 ]
  %__begin.sroa.0.01 = phi i32* [ %52, %.lr.ph ], [ %__begin.sroa.0.01.ph, %.lr.ph.preheader46 ]
  %50 = load i32, i32* %__begin.sroa.0.01, align 4, !tbaa !8
  %51 = add nsw i32 %50, %x.02
  %52 = getelementptr inbounds i32, i32* %__begin.sroa.0.01, i64 1
  %53 = icmp eq i32* %52, %4
  br i1 %53, label %._crit_edge.loopexit, label %.lr.ph, !llvm.loop !13
}
```

```
%__vector_base = type { i32*, i32*, %__compressed_pair }
%__compressed_pair = type { %__libcpp_compressed_pair_imp }
%__libcpp_compressed_pair_imp = type { i32* }

define i32 @sum(%vector* nocapture readonly dereferenceable(24) %v) {
entry:
  %begin_ptr = getelementptr inbounds %vector, %vector* %v, i64 0, i32 0, i32 0
  %begin = load i32*, i32** %begin_ptr, align 8
  %end_ptr = getelementptr inbounds %vector, %vector* %v, i64 0, i32 0, i32 1
  %end = load i32*, i32** %end_ptr, align 8
  br label %loop.head

loop.head:
  %ptr = phi i32* [ %begin, %entry ], [ %ptr.next, %loop.latch ]
  %x = phi i32 [ 0, %entry ], [ %x.next, %loop.latch ]
  %cond = icmp eq i32* %ptr, %end
  br i1 %cond, label %exit, label %loop.latch

loop.latch:
  %i = load i32, i32* %ptr, align 4
  %x.next = add nsw i32 %x, %i
  %ptr.next = getelementptr inbounds i32, i32* %ptr, i64 1
  br label %loop.head

exit:
  ret i32 %x
}
```

```
  %begin_ptr = getelementptr inbounds %vector, %vector* %v, i64 0, i32 0, i32 0
  %begin = load i32*, i32** %begin_ptr, align 8
  %end_ptr = getelementptr inbounds %vector, %vector* %v, i64 0, i32 0, i32 1
  %end = load i32*, i32** %end_ptr, align 8
  %precond = icmp eq i32* %begin, %end
  br i1 %precond, label %exit, label %loop.ph

loop.ph:                                          ; preds = %entry
  br label %loop

loop:                                             ; preds = %loop.ph, %loop
  %x = phi i32 [ 0, %loop.ph ], [ %x.next, %loop ]
  %ptr = phi i32* [ %begin, %loop.ph ], [ %ptr.next, %loop ]
  %i = load i32, i32* %ptr, align 4
  %x.next = add nsw i32 %x, %i
  %ptr.next = getelementptr inbounds i32, i32* %ptr, i64 1
  %cond = icmp eq i32* %ptr.next, %end
  br i1 %cond, label %loop.exit, label %loop

loop.exit:                                        ; preds = %loop
  %x.lcssa = phi i32 [ %x.next, %loop ]
  br label %exit

exit:                                             ; preds = %loop.exit, %entry
  %x.result = phi i32 [ %x.lcssa, %loop.exit ], [ 0, %entry ]
  ret i32 %x.result
}
```

```
entry:
  %begin_ptr = getelementptr inbounds %vector, %vector* %v, i64 0, i32 0, i32 0
  %begin = load i32*, i32** %begin_ptr, align 8
  %end = getelementptr inbounds i32, i32* %begin, i64 4
  %precond = icmp eq i32* %begin, %end
  br i1 %precond, label %exit, label %loop.ph

loop.ph:                                          ; preds = %entry
  br label %loop

loop:                                             ; preds = %loop.ph, %loop
  %x = phi i32 [ 0, %loop.ph ], [ %x.next, %loop ]
  %ptr = phi i32* [ %begin, %loop.ph ], [ %ptr.next, %loop ]
  %i = load i32, i32* %ptr, align 4
  %x.next = add nsw i32 %x, %i
  %ptr.next = getelementptr inbounds i32, i32* %ptr, i64 1
  %cond = icmp eq i32* %ptr.next, %end
  br i1 %cond, label %loop.exit, label %loop

loop.exit:                                        ; preds = %loop
  %x.lcssa = phi i32 [ %x.next, %loop ]
  br label %exit

exit:                                             ; preds = %loop.exit, %entry
  %x.result = phi i32 [ %x.lcssa, %loop.exit ], [ 0, %entry ]
  ret i32 %x.result
}
```

```
  %begin_ptr = getelementptr inbounds %vector, %vector* %v, i64 0, i32 0, i32 0
  %begin = load i32*, i32** %begin_ptr, align 8
  %end = getelementptr inbounds i32, i32* %begin, i64 4
  %precond = icmp eq i32* %begin, %end
  br i1 %precond, label %exit, label %loop.ph

loop.ph:                                          ; preds = %entry
  br label %loop

loop:                                             ; preds = %loop.ph
  %i = load i32, i32* %begin, align 4
  %ptr.next = getelementptr inbounds i32, i32* %begin, i64 1
  %i.1 = load i32, i32* %ptr.next, align 4
  %x.next.1 = add nsw i32 %i, %i.1
  %ptr.next.1 = getelementptr inbounds i32, i32* %ptr.next, i64 1
  %i.2 = load i32, i32* %ptr.next.1, align 4
  %x.next.2 = add nsw i32 %x.next.1, %i.2
  %ptr.next.2 = getelementptr inbounds i32, i32* %ptr.next.1, i64 1
  %i.3 = load i32, i32* %ptr.next.2, align 4
  %x.next.3 = add nsw i32 %x.next.2, %i.3
  %ptr.next.3 = getelementptr inbounds i32, i32* %ptr.next.2, i64 1
  br label %exit

exit:                                             ; preds = %loop, %entry
  %x.result = phi i32 [ %x.next.3, %loop ], [ 0, %entry ]
  ret i32 %x.result
}
```

```
  %precond = icmp eq i32* %begin, %end
  br i1 %precond, label %exit, label %loop.ph

loop.ph:                                          ; preds = %entry
  br label %loop

loop:                                             ; preds = %loop, %loop.ph
  %x = phi i32 [ 0, %loop.ph ], [ %x.next, %loop ]
  %ptr = phi i32* [ %begin, %loop.ph ], [ %ptr.next, %loop ]
  %i = load i32, i32* %ptr, align 4
  %x.next = add nsw i32 %x, %i
  %first = load i32, i32* %begin, align 4
  %ptr.next = getelementptr inbounds i32, i32* %ptr, i64 1
  %cond = icmp eq i32* %ptr.next, %end
  br i1 %cond, label %loop.exit, label %loop

loop.exit:                                        ; preds = %loop
  %x.lcssa = phi i32 [ %x.next, %loop ]
  %first.lcssa = phi i32 [ %first, %loop ]
  br label %exit

exit:                                             ; preds = %loop.exit, %entry
  %x.result = phi i32 [ %x.lcssa, %loop.exit ], [ 0, %entry ]
  %scale = phi i32 [ %first.lcssa, %loop.exit ], [ 1, %entry ]
  %x.scaled = mul i32 %x.result, %scale
  ret i32 %x.scaled
}
```

```
  %end = load i32*, i32** %end_ptr, align 8
  %precond = icmp eq i32* %begin, %end
  br i1 %precond, label %exit, label %loop.ph

loop.ph:                                          ; preds = %entry
  br label %loop

loop:                                             ; preds = %loop, %loop.ph
  %x = phi i32 [ 0, %loop.ph ], [ %x.next, %loop ]
  %ptr = phi i32* [ %begin, %loop.ph ], [ %ptr.next, %loop ]
  %i = load i32, i32* %ptr, align 4
  %x.next = add nsw i32 %x, %i
  %ptr.next = getelementptr inbounds i32, i32* %ptr, i64 1
  %cond = icmp eq i32* %ptr.next, %end
  br i1 %cond, label %loop.exit, label %loop

loop.exit:                                        ; preds = %loop
  %x.lcssa = phi i32 [ %x.next, %loop ]
  %first.le = load i32, i32* %begin, align 4
  br label %exit

exit:                                             ; preds = %loop.exit, %entry
  %x.result = phi i32 [ %x.lcssa, %loop.exit ], [ 0, %entry ]
  %scale = phi i32 [ %first.le, %loop.exit ], [ 1, %entry ]
  %x.scaled = mul i32 %x.result, %scale
  ret i32 %x.scaled
}
```

```
  %precond = icmp eq i32* %begin, %end
  br i1 %precond, label %exit, label %loop.ph

loop.ph:                                    ; preds = %entry
  br label %loop

loop:                                       ; preds = %loop.ph, %loop
  %x = phi i32 [ 0, %loop.ph ], [ %x.next, %loop ]
  %ptr = phi i32* [ %begin, %loop.ph ], [ %ptr.next, %loop ]
  %i = load i32, i32* %ptr, align 4
  %x.next = add nsw i32 %x, %i
  %ptr.next = getelementptr inbounds i32, i32* %ptr, i64 1
  %cond = icmp eq i32* %ptr.next, %end
  br i1 %cond, label %loop.exit, label %loop, !llvm.loop !0

loop.exit:                                  ; preds = %loop.latch
  %x.lcssa = phi i32 [ %x.next, %loop ]
  br label %exit

exit:                                       ; preds = %loop.exit, %entry
  %x.result = phi i32 [ %x.lcssa, %loop.exit ], [ 0, %entry ]
  ret i32 %x.result
}

!0 = distinct !{!0, !1, !2}
!1 = !{!"llvm.loop.vectorize.width", i32 1}
!2 = !{!"llvm.loop.interleave.count", i32 2}
```

```
loop.ph:                                          ; preds = %entry
  %begin2 = ptrtoint i32* %begin to i64
  %scevgep = getelementptr i32, i32* %end, i64 -1
  %0 = ptrtoint i32* %scevgep to i64
  %1 = sub i64 %0, %begin2
  %2 = lshr i64 %1, 2
  %3 = add nuw nsw i64 %2, 1
  %min.iters.check = icmp ult i64 %3, 2
  br i1 %min.iters.check, label %scalar.ph, label %min.iters.checked

min.iters.checked:                                ; preds = %loop.ph
  %n.vec = and i64 %3, 9223372036854775806
  %cmp.zero = icmp eq i64 %n.vec, 0
  %ind.end = getelementptr i32, i32* %begin, i64 %n.vec
  br i1 %cmp.zero, label %scalar.ph, label %vector.ph

vector.ph:                                        ; preds = %min.iters.checked
  br label %vector.body

vector.body:                                      ; preds = %vector.body, %vector.ph
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ]
  %vec.phi = phi i32 [ 0, %vector.ph ], [ %7, %vector.body ]
  %vec.phi4 = phi i32 [ 0, %vector.ph ], [ %8, %vector.body ]
  %next.gep = getelementptr i32, i32* %begin, i64 %index
  %4 = or i64 %index, 1
  %next.gep5 = getelementptr i32, i32* %begin, i64 %4
```

```
vector.ph:                                        ; preds = %min.iters.checked
  br label %vector.body

vector.body:                                      ; preds = %vector.body, %vector.ph
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ]
  %vec.phi = phi i32 [ 0, %vector.ph ], [ %7, %vector.body ]
  %vec.phi4 = phi i32 [ 0, %vector.ph ], [ %8, %vector.body ]
  %next.gep = getelementptr i32, i32* %begin, i64 %index
  %4 = or i64 %index, 1
  %next.gep5 = getelementptr i32, i32* %begin, i64 %4
  %5 = load i32, i32* %next.gep, align 4
  %6 = load i32, i32* %next.gep5, align 4
  %7 = add nsw i32 %vec.phi, %5
  %8 = add nsw i32 %vec.phi4, %6
  %index.next = add i64 %index, 2
  %9 = icmp eq i64 %index.next, %n.vec
  br i1 %9, label %middle.block, label %vector.body, !llvm.loop !0

middle.block:                                     ; preds = %vector.body
  %bin.rdx = add i32 %8, %7
  %cmp.n = icmp eq i64 %3, %n.vec
  br i1 %cmp.n, label %loop.exit, label %scalar.ph

scalar.ph:                                        ; preds = %middle.block, %min.iters.checked,
  %bc.resume.val = phi i32* [ %ind.end, %middle.block ], [ %begin, %loop.ph ], [ %begin, %min.
  %bc.merge.rdx = phi i32 [ %bin.rdx, %middle.block ], [ 0, %loop.ph ], [ 0, %min.iters.checke
  br label %loop
```

WHAT HAPPENS WHEN THESE ABSTRACTIONS ARE COMBINED?

```
int f(int a, int b) {
  int c;
  g(a, b, c);
  return a + b + c;
}

void g(int a, int b, int &c) {
  c = a * b;
}
```

```c
int f(int a, int b) {
  int c = g(a, b);
  return a + b + c;
}

int g(int a, int b) {
  return a * b;
}
```

```cpp
struct S {
  float x, y, z;
  double delta;

  double compute();
};
```

```cpp
double f() {
  S s;
  s.x = /* expensive compute */;
  s.y = /* expensive compute */;
  s.z = /* expensive compute */;
  s.delta = s.x - s.y - s.z;
  return s.compute();
}
```

```c
struct S {
  float x, y, z;
  double delta;
};

double compute(S s);
```

```c
double f() {
  S s;
  s.x = /* expensive compute */;
  s.y = /* expensive compute */;
  s.z = /* expensive compute */;
  if (s.x - s.y - s.z < C)
    s.delta = C;
  else
    s.delta = s.x - s.y - s.z;
  return compute(s);
}
```

IN CONCLUSION?

QUESTIONS!
(MAYBE ANSWERS?)