C++ COROUTINES: UNDER THE COVERS

AMAZING COROUTINE DISAPPEARING ACT

Microsoft Visual C++ Team
Gor Nishanov

Ubuntu LTS 16.04
clang 4.0 (forked)
llvm 4.0 (trunk)

COMPILER


```
generator<int> seq(int start) {
  for (;;)
    co_yield start++;
}
```

```
define void @seq(%struct.generator* noalias sret %agg.result) #0 {
entry:
  %coro.promise = alloca %"struct.generator<int>::promise_type", align 4
  %coro.gro = alloca %struct.generator, align 8
  %ref.tmp = alloca %"struct.std::suspend_always", align 1
  %undef.agg.tmp = alloca %"struct.std::suspend_always", align 1
  %agg.tmp = alloca %"struct.std::coroutine_handle.0", align 8
                    ...
```

```
seq:
    pushq   %rbx
    movq    %rdi, %rbx
    movl    $32, %edi
    callq   _Znwm@PLT
      ...
```

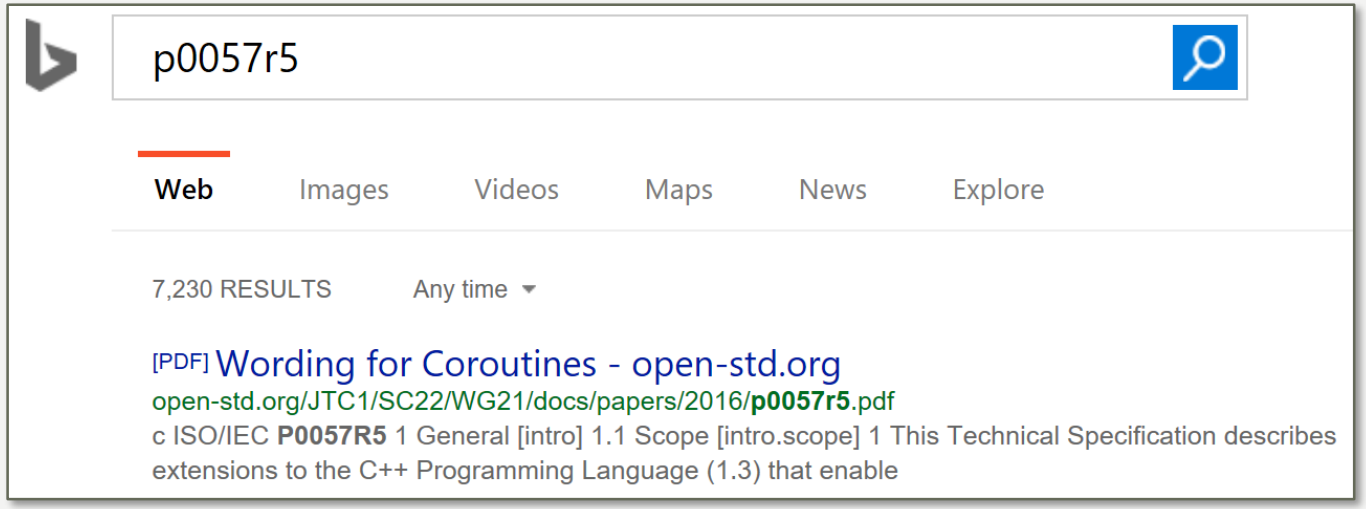
TWO KIND OF COROUTINES

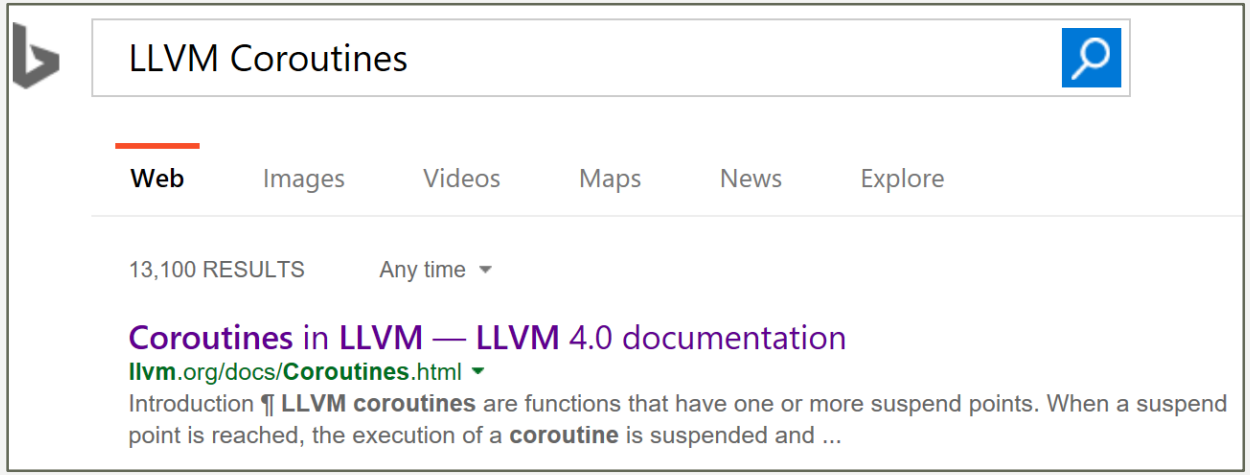
C++ Coroutines:
- co_await,
- co_yield

LLVM Coroutines:

- llvm.coro.begin
- llvm.coro.suspend
- llvm.coro.end

- llvm.coro.resume
- llvm.coro.destroy





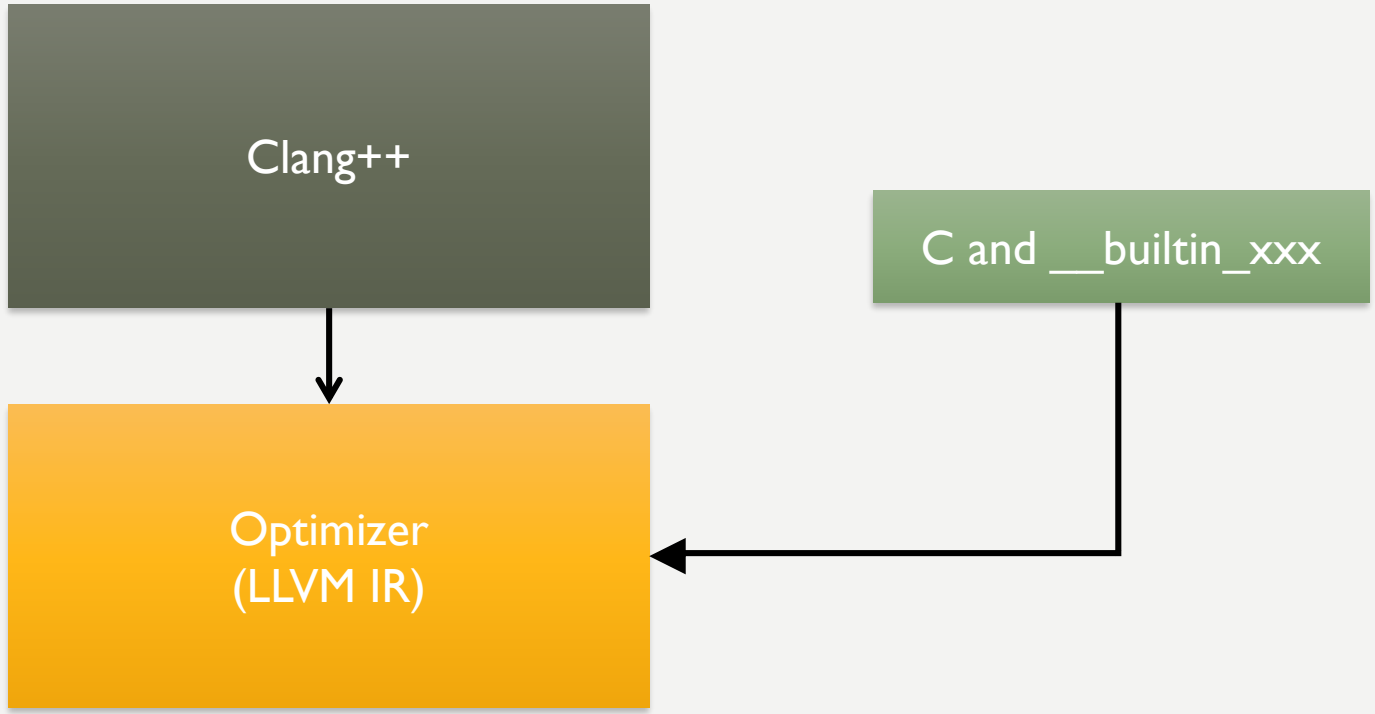
TWO KIND OF COROUTINES

C++ Coroutines:
co_await,
co_yield

LLVM Coroutines:

llvm.coro.begin
llvm.coro.suspend
llvm.coro.end

llvm.coro.resume
llvm.coro.destroy



COROUTINES IN C

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);

  for (int i = n;; ++i) {
    CORO_SUSPEND(hdl);
    print(i);
    CORO_SUSPEND(hdl);
    print(-i);
  }

  CORO_END(hdl, free);
}
```

1, -1, 2, -2, 3, -3, …

```
int main() {
  void* coro = f(1);
  for (int i = 0; i < 4; ++i) {
    CORO_RESUME(coro);
  }
  CORO_DESTROY(coro);
}
```

```
define i32 @main() {
  call void @print(i32 1)
  call void @print(i32 -1)
  call void @print(i32 2)
  call void @print(i32 -2)
  ret i32 0
}
```

BUILD COROUTINE FRAME

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);

  for (int i = n;; ++i) {

    CORO_SUSPEND(hdl);
    print(i);

    CORO_SUSPEND(hdl);
    print(-i);
  }

  CORO_END(hdl, free);
}
```

```
struct f.frame {
  int i;
};
```

1, -1, 2, -2, 3, -3, …

REWRITE ACCESS TO SPILLED VARIABLES

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);

  for (int i = n;; ++i) {

    CORO_SUSPEND(hdl);
    print(i);

    CORO_SUSPEND(hdl);
    print(-i);
  }

  CORO_END(hdl, free);
}
```

```
struct f.frame {
  int i;
};
```

REWRITE ACCESS TO SPILLED VARIABLES

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  for (frame->i = n;; ++frame->i) {

    CORO_SUSPEND(hdl);
    print(frame->i);

    CORO_SUSPEND(hdl);
    print(-frame->i);
  }

  CORO_END(hdl, free);
}
```

```
struct f.frame {
  int i;
};
```

CREATE JUMP POINTS

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  for (frame->i = n;; ++frame->i) {

    CORO_SUSPEND(hdl);
    print(frame->i);

    CORO_SUSPEND(hdl);
    print(-frame->i);
  }

  CORO_END(hdl, free);
}
```

```
struct f.frame {
  int i;
};
```

CREATE JUMP POINTS

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  for (frame->i = n;; ++frame->i) {

    CORO_SUSPEND(hdl);
    print(frame->i);

    CORO_SUSPEND(hdl);
    print(-frame->i);
  }

  CORO_END(hdl, free);
}
```

```
struct f.frame {
  int suspend_index;
  int i;
};
```

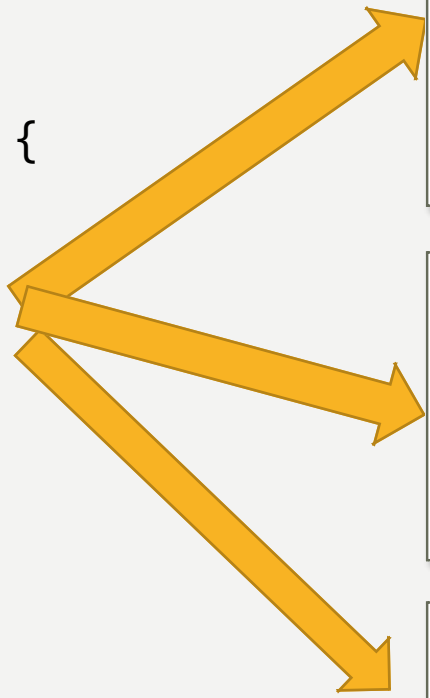
CREATE JUMP POINTS

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  for (frame->i = n;; ++frame->i) {
    frame->suspend_index = 0;
r0: CORO_SUSPEND(hdl);
    print(frame->i);
    frame->suspend_index = 1;
r1: CORO_SUSPEND(hdl);
    print(-frame->i);
  }

  CORO_END(hdl, free);
}
```

```
struct f.frame {
  int suspend_index;
  int i;
};
```

SPLIT COROUTINE

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  for (frame->i = n;; ++frame->i) {
    frame->suspend_index = 0;
r0: CORO_SUSPEND(hdl);
    print(frame->i);
    frame->suspend_index = 1;
r1: CORO_SUSPEND(hdl);
    print(-frame->i);
  }

  CORO_END(hdl, free);
}
```

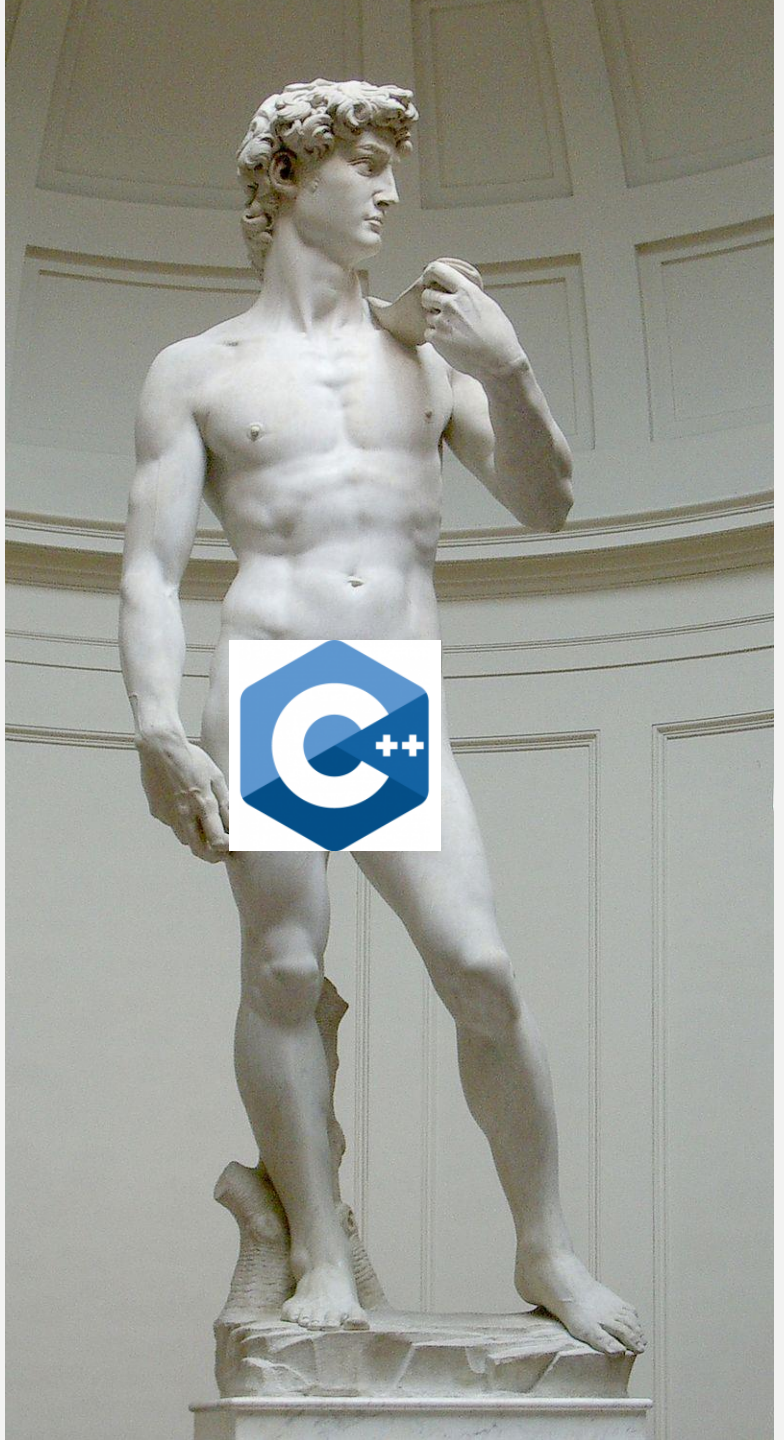


```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);
  …
  return hdl;
}
```
Coroutine Start Function

```
void f.resume(f.frame* frame) {
 switch(frame->suspend_index){
   …
 }
}
```
Coroutine Resume Function

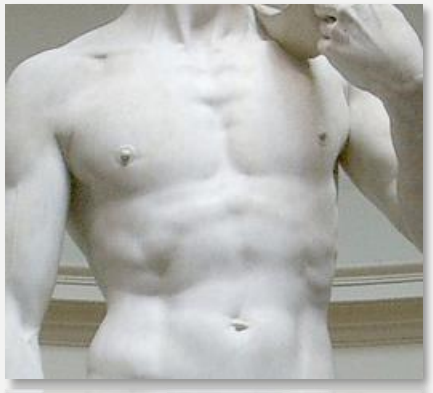
```
void f.destroy(f.frame* frame) {
 switch(frame->suspend_index){
   …
 }
 free(frame);
}
```
Coroutine Destroy Function

BEAUTY IS PURGATION OF SUPERFLUITIES



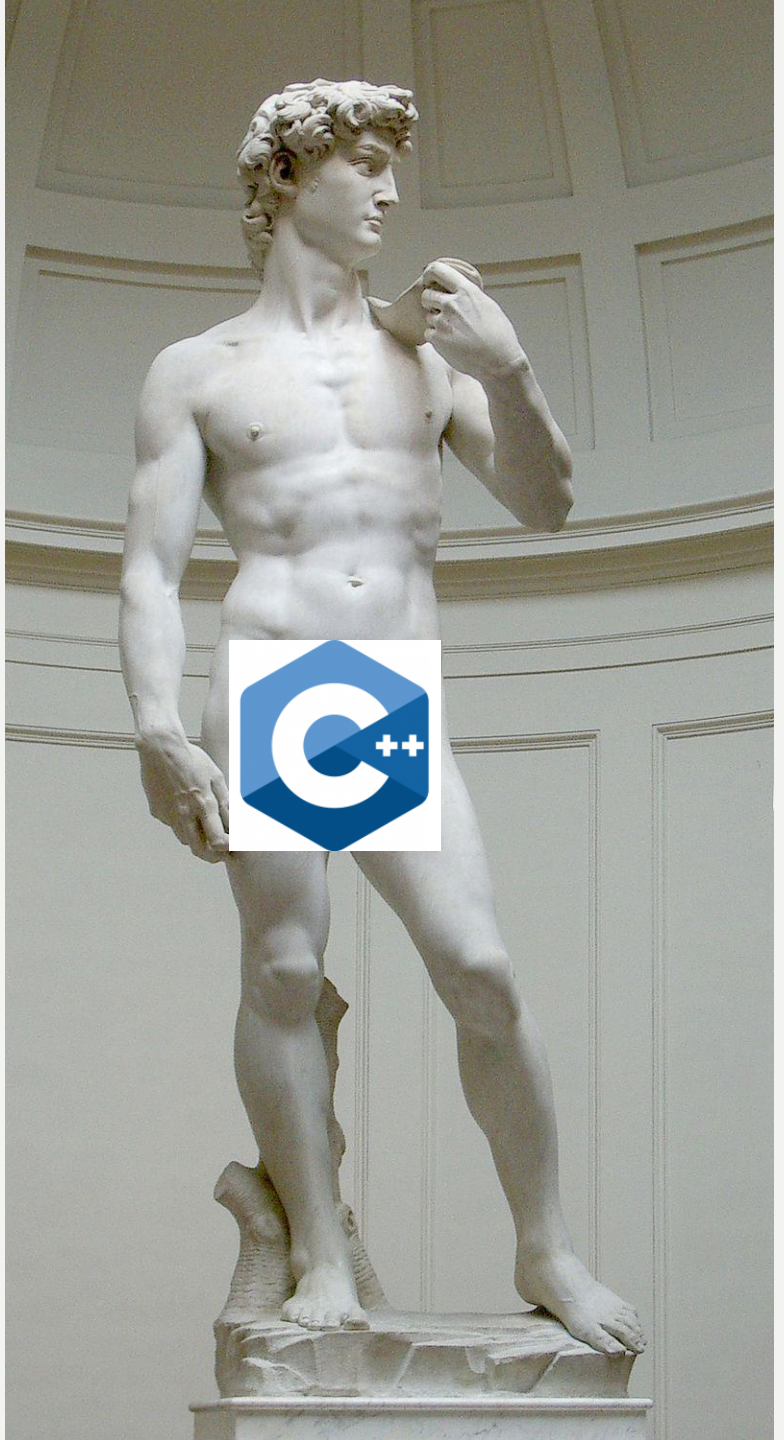
BEAUTY IS PURGATION OF SUPERFLUITIES

Coroutine Start Function



Coroutine Destroy Function

Coroutine Resume Function



CREATE RESUME CLONE

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;

  for (frame->i = n;; ++frame->i) {
    frame->suspend_index = 0;
r0: CORO_SUSPEND(hdl);
    print(frame->i);
    frame->suspend_index = 1;
r1: CORO_SUSPEND(hdl);
    print(-frame->i);
  }

  CORO_END(hdl, free);
}
```

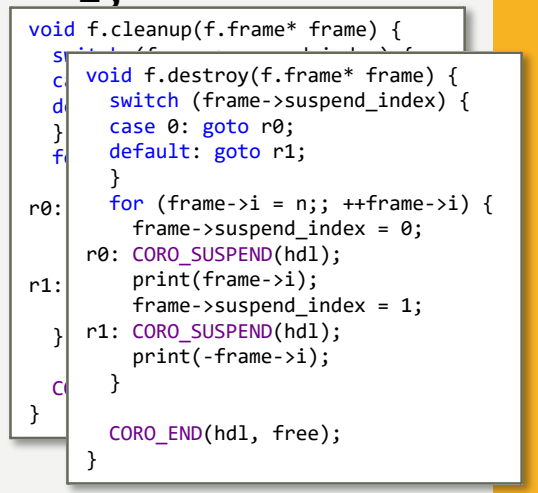
Clone

```
void* f'(int n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;

  for (frame->i = n;; ++frame->i) {
    frame->suspend_index = 0;
r0: CORO_SUSPEND(hdl);
    print(frame->i);
    frame->suspend_index = 1;
r1: CORO_SUSPEND(hdl);
    print(-frame->i);
  }

  CORO_END(hdl, free);
}
```

CREATE RESUME CLONE

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;

  for (frame->i = n;; ++frame->i) {
    frame->suspend_index = 0;
r0: CORO_SUSPEND(hdl);
    print(frame->i);
    frame->suspend_index = 1;
r1: CORO_SUSPEND(hdl);
    print(-frame->i);
  }

  CORO_END(hdl, free);
}
```

```
void f.resume(f.frame* frame) {
  switch (frame->suspend_index) {
  case 0: goto r0;
  default: goto r1;
  }
  for (frame->i = n;; ++frame->i) {
    frame->suspend_index = 0;
r0: CORO_SUSPEND(hdl);
    print(frame->i);
    frame->suspend_index = 1;
r1: CORO_SUSPEND(hdl);
    print(-frame->i);
  }

  CORO_END(hdl, free);
}
```



PURGATION OF SUPERFLUITIES

```
#define CORO_SUSPEND()                          \
  switch (__builtin_coro_suspend()) {           \
  case -1:                                      \
    goto coro_Suspend;                          \
  case 1:                                       \
    goto coro_Cleanup;                          \
  default:                                      \
    break;                                      \
  }
```

```
#define CORO_END(hdl, FreeFunc)                 \
coro_Cleanup : {                                \
    void *coro_mem = __builtin_coro_free(hdl);\
    if (coro_mem)                               \
      FreeFunc(coro_mem);                       \
  }                                             \
coro_Suspend:                                   \
  __builtin_coro_end();                         \
  return coro_hdl;
```

__builtin_coro_suspend()	
-1	In start function
0	In resume function
1	In destroy and cleanup functions

__builtin_coro_free()	
0	In cleanup function
hdl	elsewhere

AFTER CLEANUP

```
void* f(int *n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = n;
  frame->suspend_index = 0;
  return coro_hdl;
}
```

```
void f.destroy(f.frame* frame) {
  free(frame);
}
```

```
void f.cleanup(f.frame* frame) {}
```

```
struct f.frame {
  FnPtr ResumeFn;
  FnPtr DestroyFn;
  int suspend_index;
  int i;
};
```

```
void f.resume(f.frame* frame) {
  if (frame->index == 0) {
    print(frame->i);
    frame->suspend_index = 1;
  }
  else {
    print(-frame->i);
    ++frame->i;
    frame->suspend_index = 0;
  }
}
```

OPTIMIZING COROUTINES

INLINING

```
int main() {
  void* coro = f(1);
  CORO_RESUME(coro);
  CORO_RESUME(coro);
  CORO_DESTROY(coro);
}
```

```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = n;
  frame->suspend_index = 0;
  return coro_hdl;
}
```

```
int main() {
  void* coro = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)coro;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = 1;
  frame->suspend_index = 0;
  CORO_RESUME(coro);
  CORO_RESUME(coro);
  CORO_DESTROY(coro);
}
```

DEVIRTUALIZATION

```
int main() {
  void* coro = f();
  CORO_RESUME(coro);
  CORO_RESUME(coro);
  CORO_DESTROY(coro);
}
```

```
void* f(int *n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = n;
  frame->suspend_index = 0;
  return coro_hdl;
}
```

```
int main() {
  void* coro = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)coro;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = 1;
  frame->suspend_index = 0;
  CORO_RESUME(coro);
  CORO_RESUME(coro);
  CORO_DESTROY(coro);
}
```

DEVIRTUALIZATION

```
int main() {
  void* coro = f();
  CORO_RESUME(coro);
  CORO_RESUME(coro);
  CORO_DESTROY(coro);
}
```

```
void* f(int *n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = n;
  frame->suspend_index = 0;
  return coro_hdl;
}
```

```
int main() {
  void* coro = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)coro;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = 1;
  frame->suspend_index = 0;
  f.resume(coro);
  f.resume(coro);
  f.destroy(coro);
}
```

HEAP ALLOCATION ELISION

```
int main() {
  void* coro = f();
  CORO_RESUME(coro);
  CORO_RESUME(coro);
  CORO_DESTROY(coro);
}
```

```
void* f(int *n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = n;
  frame->suspend_index = 0;
  return coro_hdl;
}
```

```
int main() {
  void* coro = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)coro;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = 1;
  frame->suspend_index = 0;
  f.resume(coro);
  f.resume(coro);
  f.destroy(coro);
}
```

HEAP ALLOCATION ELISION

```
int main() {
  void* coro = f();
  CORO_RESUME(coro);
  CORO_RESUME(coro);
  CORO_DESTROY(coro);
}
```

```
void* f(int *n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = n;
  frame->suspend_index = 0;
  return coro_hdl;
}
```

```
int main() {
  void* coro = alloca(sizeof(f.frame));
  f.frame* frame = (f.frame*)coro;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = n;
  frame->suspend_index = 0;
  f.resume(coro);
  f.resume(coro);
  f.cleanup(coro);
}
```

NON-COROUTINE RELATED OPTIMIZATIONS FINISH THE JOB

MORE INLINING

```
int main() {
  void* coro = alloca(sizeof(f.frame));
  f.frame* frame = (f.frame*)coro;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = n;
  frame->suspend_index = 0;
  f.resume(coro);
  f.resume(coro);
  f.cleanup(coro);
}
void f.resume(f.frame* frame) {
  if (frame->index == 0) {
    print(frame->i);
    frame->suspend_index = 1;
  }
  else {
    print(-frame->i);
    ++frame->i;
    frame->suspend_index = 0;
  }
}
```

```
int main() {
  void* coro = alloca(sizeof(f.frame));
  f.frame* frame = (f.frame*)coro;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = 1;
  frame->suspend_index = 0;
  if (frame->suspend_index == 0) {
    print(frame->i);
    frame->suspend_index = 1;
  }
  else {
    print(-frame->i);
    ++frame->i;
    frame->suspend_index = 0;
  }
 if (frame->suspend_index == 0) {
   print(frame->i);
   frame->suspend_index = 1;
 }
 else {
   print(-frame->i);
   ++frame->i;
   frame->suspend_index = 0;
 }
}
```

SROA
SCALAR REPLACEMENT OF AGGREGATES

```
int f() {
  struct Point {
    int X;
    int Y;
  };
  Point p;
  p.X = 1;
  p.Y = 2;
  return p.X + p.Y;
}
```

p_addr = &p
x_addr = p_addr + offset of Point::X
STORE 1, x_addr

y.addr = p.addr + offset of Point::Y
%y = LOAD y.addr

SROA
SCALAR REPLACEMENT OF AGGREGATES AND SSA

```
int f() {
  struct Point {
    int X;
    int Y;
  };
  Point p;
  const int p_X = 1;
  const int p_Y = 2;
  return p_X + p_Y;
}
```

SSA
COMPLICATION 1

```
int X = 1;
X = X + 1;
print(X);
```

```
const int X = 1;
const int X1 = X + 1;
print(X1)
```

SSA
COMPLICATION 2

```
entry:
  int X = 1;
loop:
  X = X + 1;
  print(X);
  goto loop;
```

```
entry:
  const int X = 1;
loop:
  const int X1 = X? + 1;
  print(X1);
  goto X;
```

$X_? = PHI(entry: X, loop: X_1)$

SROA
SCALAR REPLACEMENT OF AGGREGATES

```
int main() {
  void* coro = alloca(sizeof(f.frame));
  f.frame* frame = (f.frame*)coro;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = 1;
  frame->suspend_index = 0;
  if (frame->suspend_index == 0) {
    print(frame->i);
    frame->suspend_index = 1;
  }
  else {
    print(-frame->i);
    ++frame->i;
    frame->suspend_index = 0;
  }
  if (frame->suspend_index == 0) {
    print(frame->i);
    frame->suspend_index = 1;
  }
  else {
    print(-frame->i);
    ++frame->i;
    frame->suspend_index = 0;
  }
}
```

```
int main() {
  const auto* ResumeFn = &f.resume;
  const auto* DestroyFn = &f.destroy;
  const int i = 1;
  const int suspend_index = 0;
  if (suspend_index == 0)
bb1: print(i);
  else
bb2: print(-i);
  const int i1 = FROM(bb1) ? i : i + 1;
  const int suspend_index1 =
    FROM(bb1) ? 1 : 0;

  if (suspend_index1 == 0)
bb3: print(i1);
  else
bb4: print(-i1);
  const int i2 = FROM(bb3) ? i1 : i1 + 1;
  const int suspend_index2 =
    FROM(bb3) ? 1 : 0;
}
```

DEAD CODE ELIMINATION

```
int main() {
  const auto* ResumeFn = &f.resume;
  const auto* DestroyFn = &f.destroy;
  const int i = 1;
  const int suspend_index = 0;
  if (suspend_index == 0)
bb1: print(i);
  else
bb2: print(-i);
  const int i1 = FROM(bb1) ? i : i + 1;
  const int suspend_index1 =
    FROM(bb1) ? 1 : 0;

  if (suspend_index1 == 0)
bb3: print(i1);
  else
bb4: print(-i1);
  const int i2 = FROM(bb3) ? i1 : i1 + 1;
  const int suspend_index2 =
    FROM(bb3) ? 1 : 0;
}
```

DEAD CODE ELIMINATION

```
int main() {


  const int i = 1;
  const int suspend_index = 0;
  if (suspend_index == 0)
bb1: print(i);
  else
bb2: print(-i);
  const int i1 = FROM(bb1) ? i : i + 1;
  const int suspend_index1 = FROM(bb1) ? 1 : 0;

  if (suspend_index1 == 0)
bb3: print(i1);
  else
bb4: print(-i1);


}
```

CONSTANT PROPAGATION

```
int main() {


  const int i = 1;
  const int suspend_index = 0;
  if (suspend_index == 0)
bb1: print(i);
  else
bb2: print(-i);
  const int i1 = FROM(bb1) ? i : i + 1;
  const int suspend_index1 = FROM(bb1) ? 1 : 0;

  if (suspend_index1 == 0)
bb3: print(i1);
  else
bb4: print(-i1);


}
```

CONSTANT PROPAGATION

```
int main() {


  const int i = 1;
  const int suspend_index = 0;
  if (suspend_index == 0)
bb1: print(i);
  else
bb2: print(-i);
  const int i1 = FROM(bb1) ? i : i + 1;
  const int suspend_index1 = FROM(bb1) ? 1 : 0;

  if (suspend_index1 == 0)
bb3: print(i1);
  else
bb4: print(-i1);


}
```

CONSTANT PROPAGATION

```
int main() {




  if (0 == 0)
bb1: print(1);
  else
bb2: print(-1);
  const int i1 = FROM(bb1) ? 1 : 1 + 1;
  const int suspend_index1 = FROM(bb1) ? 1 : 0;

  if (suspend_index1 == 0)
bb3: print(i1);
  else
bb4: print(-i1);


}
```

SIMPLIFY CONTROL FLOW

```
int main() {




bb1: print(1);


  const int i1 = 1;
  const int suspend_index1 = 1;

  if (suspend_index1 == 0)
bb3: print(i1);
  else
bb4: print(-i1);


}
```

ONE MORE TIME

```
int main() {



bb1: print(1);



bb4: print(-1);

}
```

WHAT ABOUT ASYNC COROUTINES?

COMPILER


```
generator<int> seq(int start) {
  for (;;)
    co_yield start++;
}
```

```
define void @seq(%struct.generator* noalias sret %agg.result) #0 {
entry:
  %coro.promise = alloca %"struct.generator<int>::promise_type", align 4
  %coro.gro = alloca %struct.generator, align 8
  %ref.tmp = alloca %"struct.std::suspend_always", align 1
  %undef.agg.tmp = alloca %"struct.std::suspend_always", align 1
  %agg.tmp = alloca %"struct.std::coroutine_handle.0", align 8
                    ...
```

```
seq:
    pushq   %rbx
    movq    %rdi, %rbx
    movl    $32, %edi
    callq   _Znwm@PLT
      ...
```

ZOOMING INTO AN OPTIMIZER

Early Passes:

-simplifycfg –domtree
-sroa -early-cse
-memoryssa -gvn-hoist

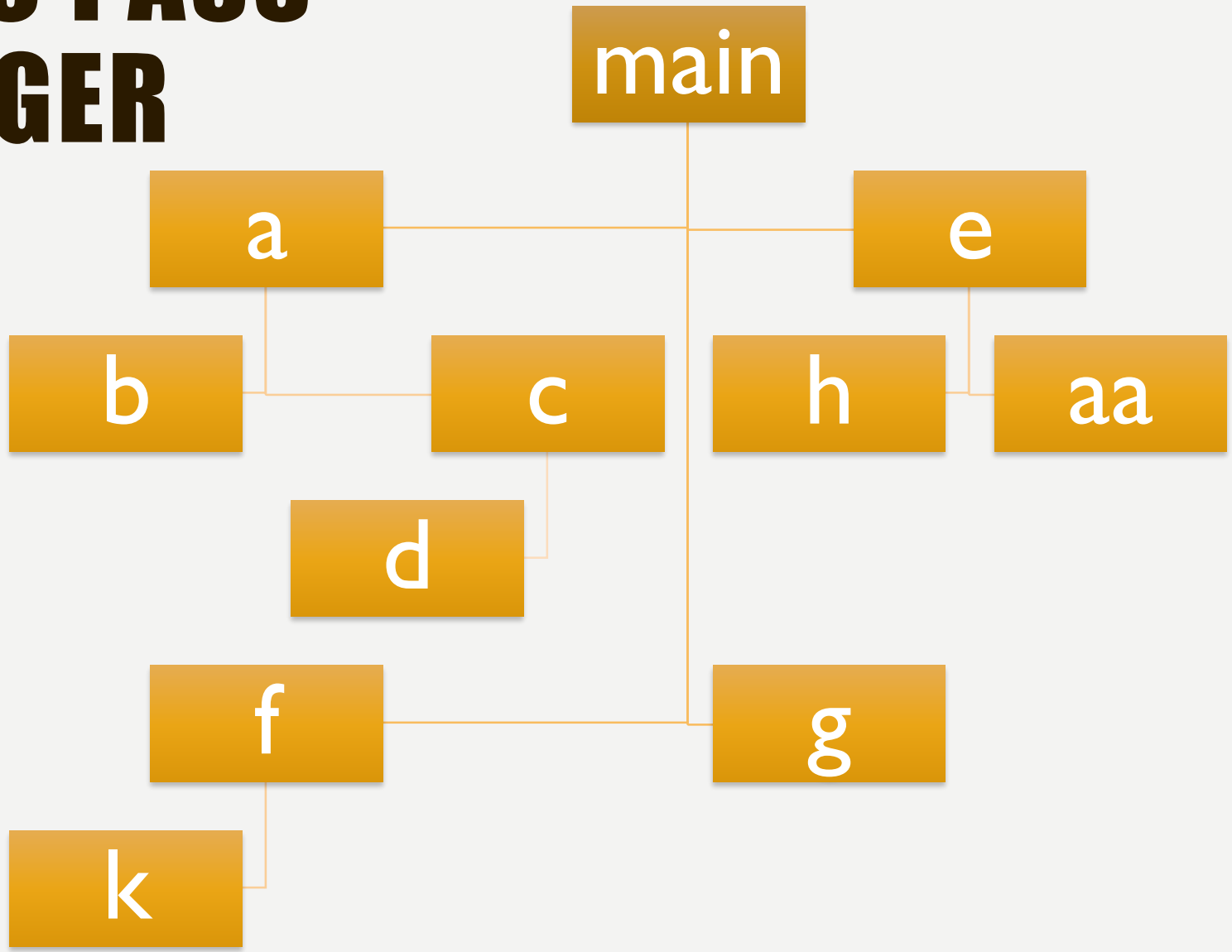
CGSCC PM

-forceattrs -inferattrs -ipsccp -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa -aa -instcombine -simplifycfg -pgo-icall-prom -basiccg -globals-aa -prune-eh -inline -functionattrs -coro-split -domtree -sroa -early-cse -speculative-execution -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -domtree -basicaa -aa -instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -basicaa -aa -scalar-evolution -loop-rotate -licm -loop-unswitch -simplifycfg -domtree -basicaa -aa -instcombine -loops -loop-simplify -lcssa -scalar-evolution -indvars -loop-idiom -loop-deletion -loop-unroll -mldst-motion -aa -memdep -gvn -basicaa -aa -memdep -memcpyopt -sccp -domtree -demanded-bits -bdce -basicaa -aa -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -basicaa -aa -memdep -dse -loops -loop-simplify -lcssa -aa -scalar-evolution -licm -coro-elide -postdomtree -adce -simplifycfg -domtree -basicaa -aa -instcombine

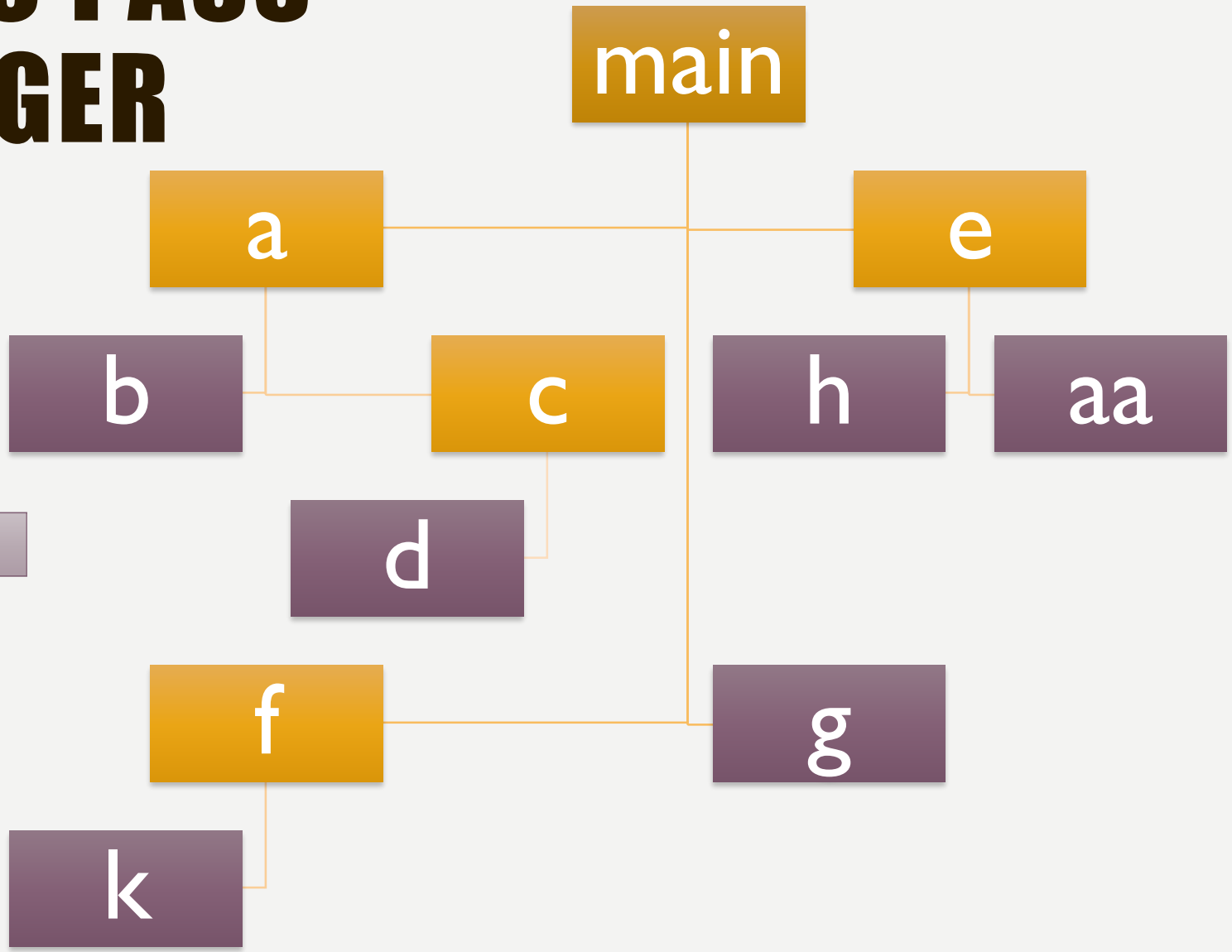
Late Passes:

-elim-avail-extern -basiccg -rpo-functionattrs -globals-aa -float2int -domtree -loops -loop-simplify -lcssa -basicaa -aa -scalar-evolution -loop-rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distribute -loop-simplify -lcssa -branch-prob -block-freq -scalar-evolution -basicaa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolution -aa -loop-accesses -loop-load-elim -basicaa -aa -instcombine -scalar-evolution -demanded-bits -slp-vectorizer -simplifycfg -domtree -basicaa -aa -instcombine -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -instcombine -loop-simplify -lcssa -scalar-evolution -licm -instsimplify -scalar-evolution -alignment-from-assumptions -strip-dead-prototypes -globaldce -constmerge -coro-cleanup

CGSCC PASS MANAGER

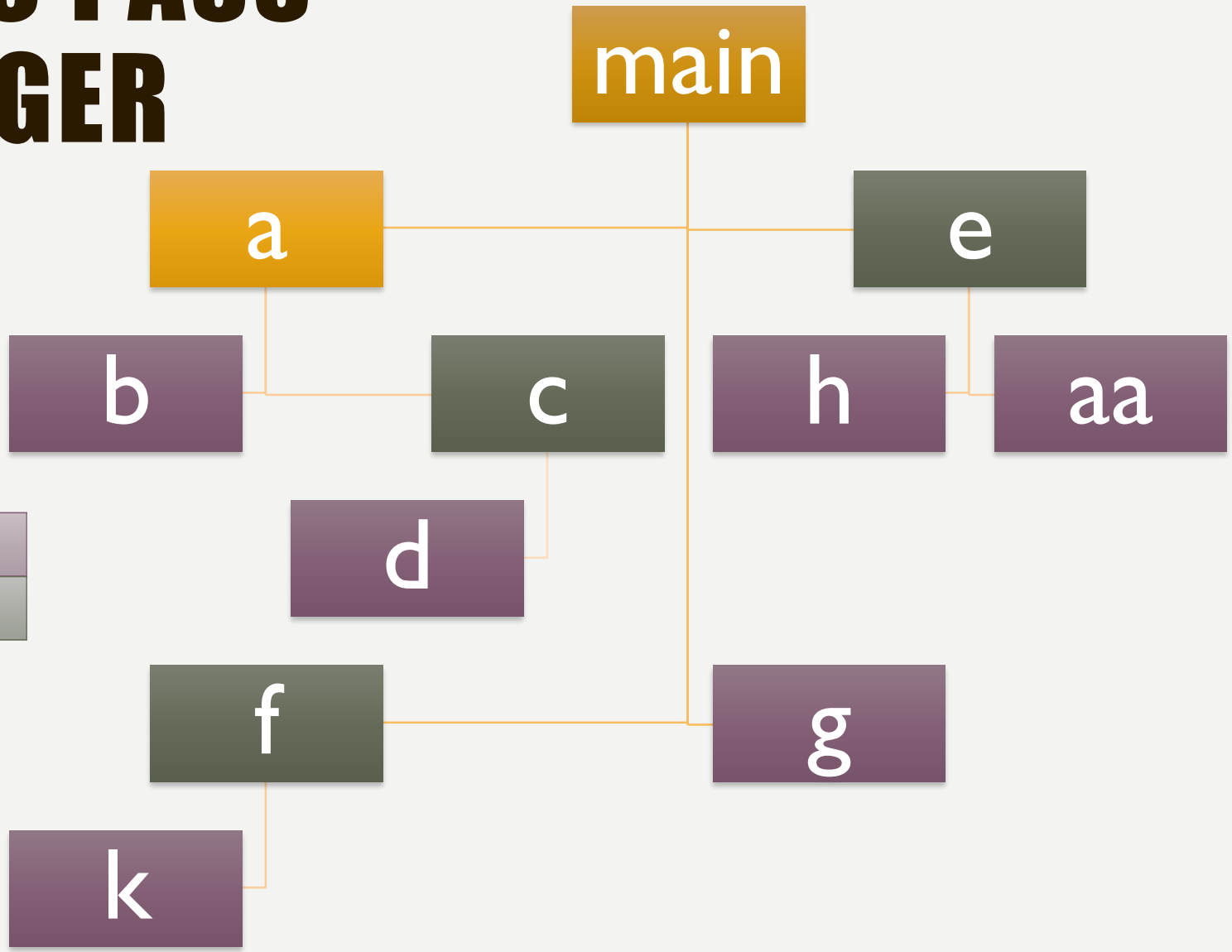


CGSCC PASS MANAGER



Step 1: Optimize purple

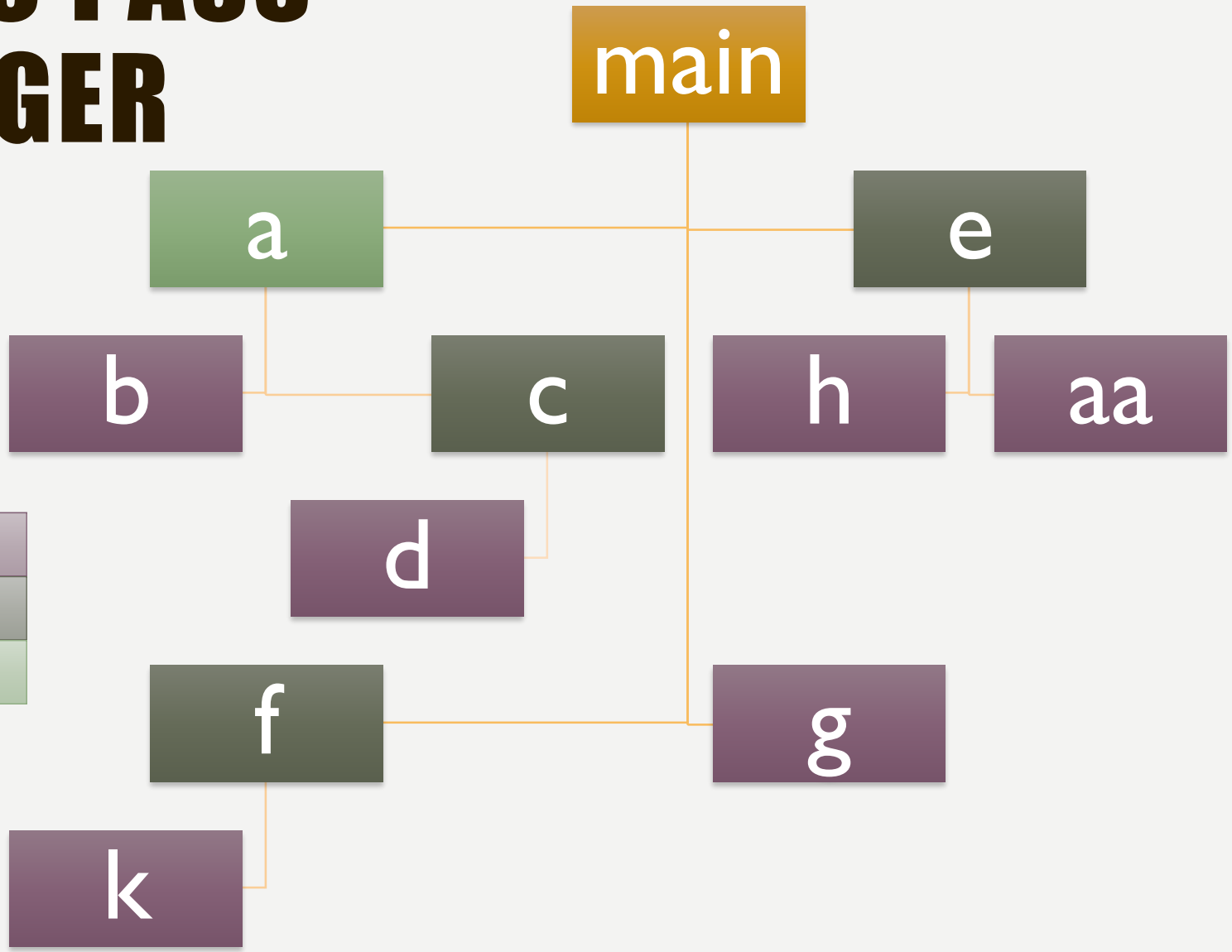
CGSCC PASS MANAGER



Step 1: Optimize purple

Step 2: Optimize gray

CGSCC PASS MANAGER

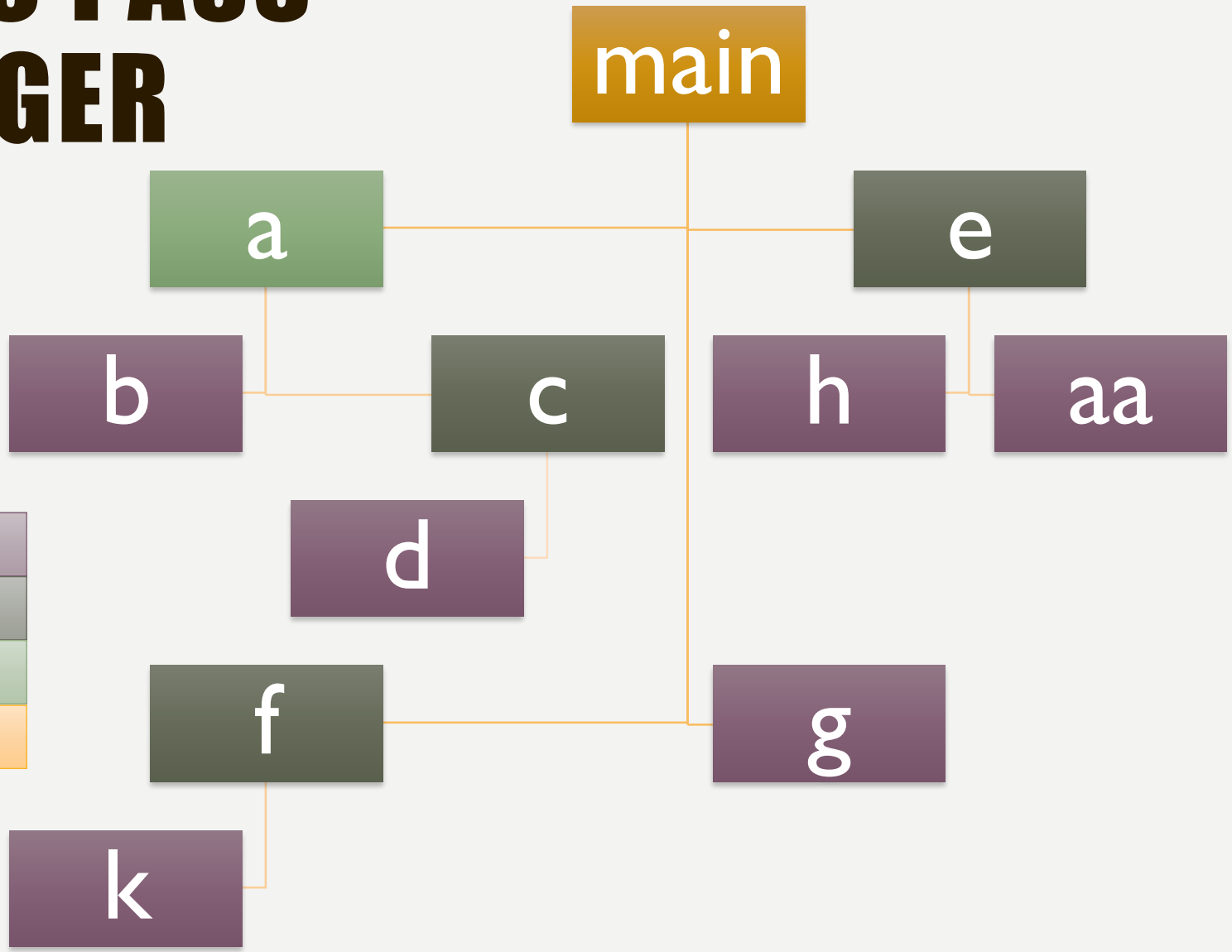


Step 1: Optimize purple

Step 2: Optimize gray

Step 3: Optimize green

CGSCC PASS MANAGER



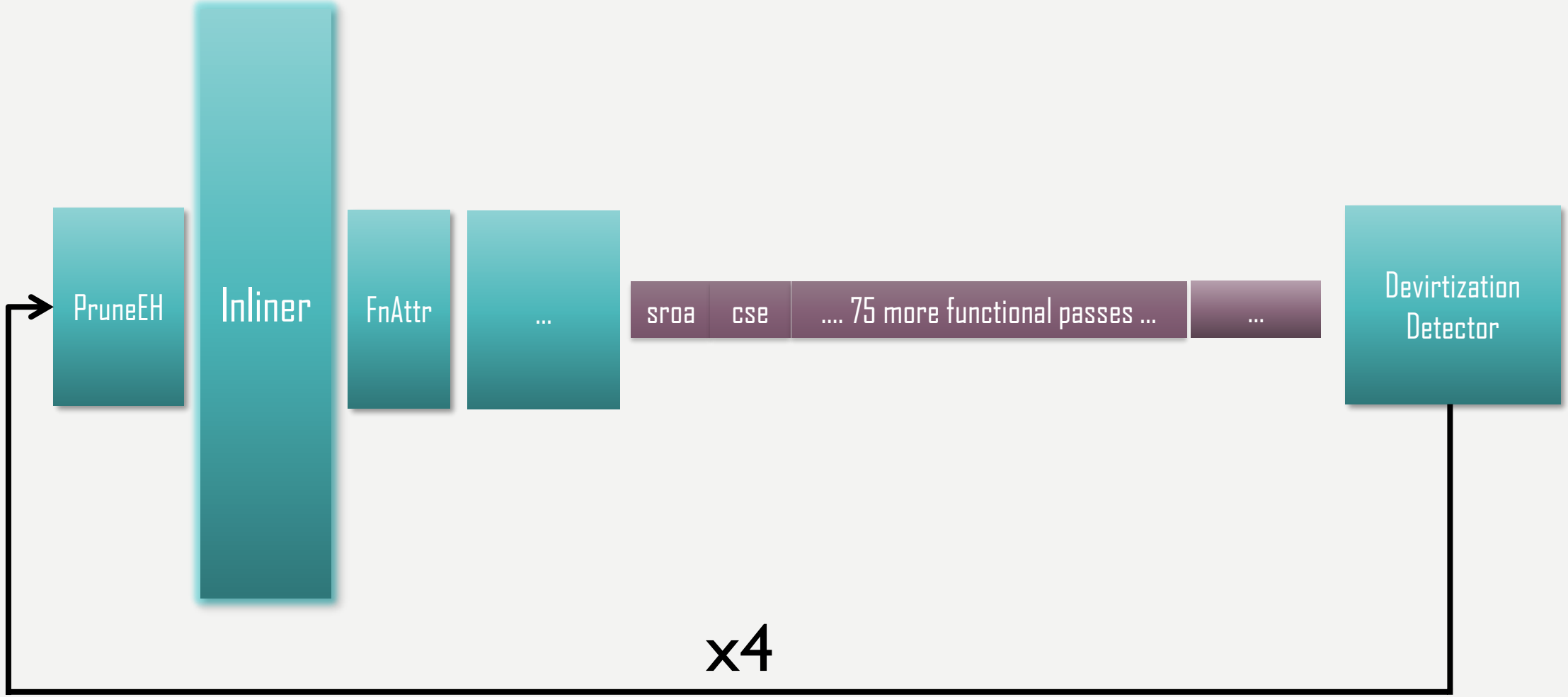
Step 1: Optimize purple

Step 2: Optimize gray

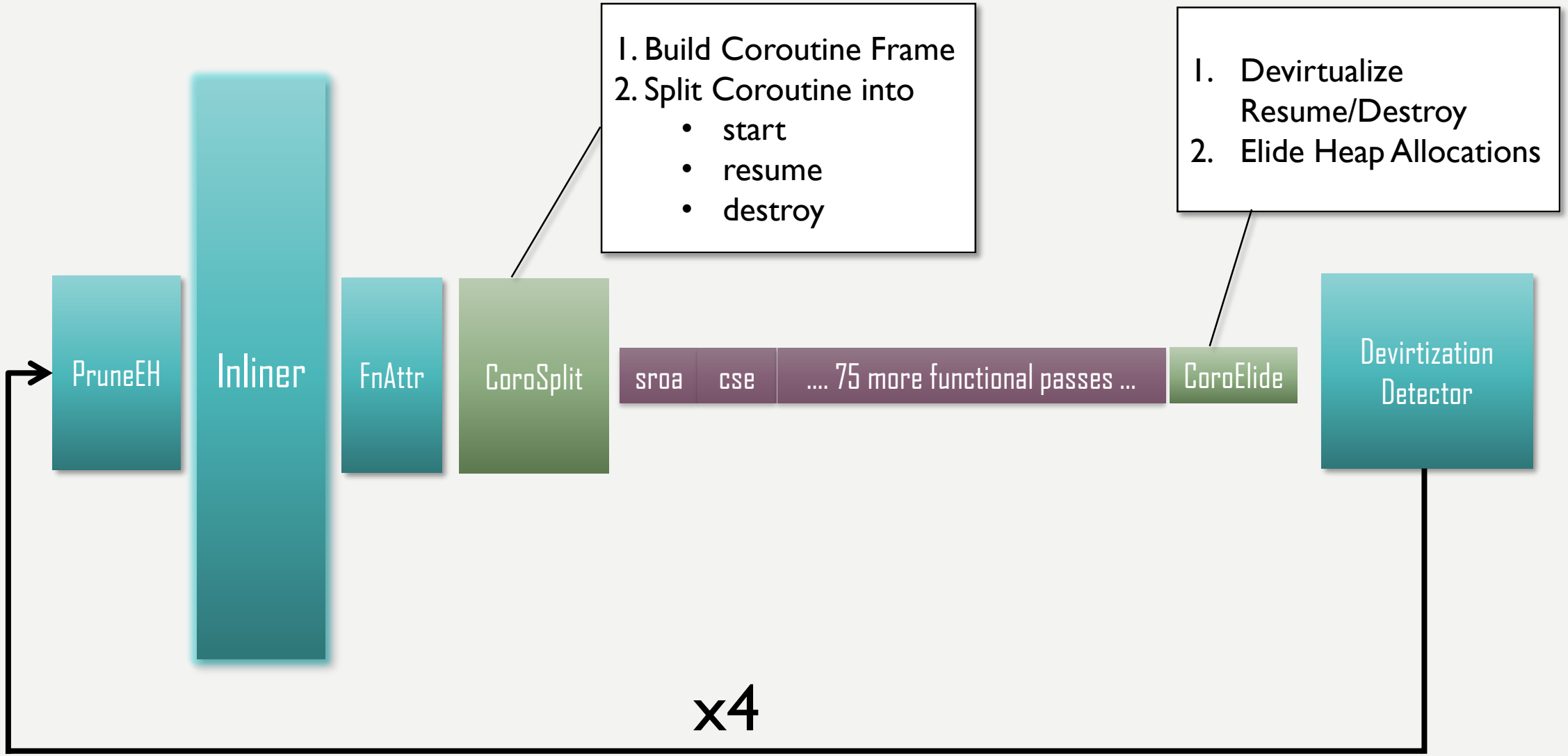
Step 3: Optimize green

Step 4: Optimize yellow

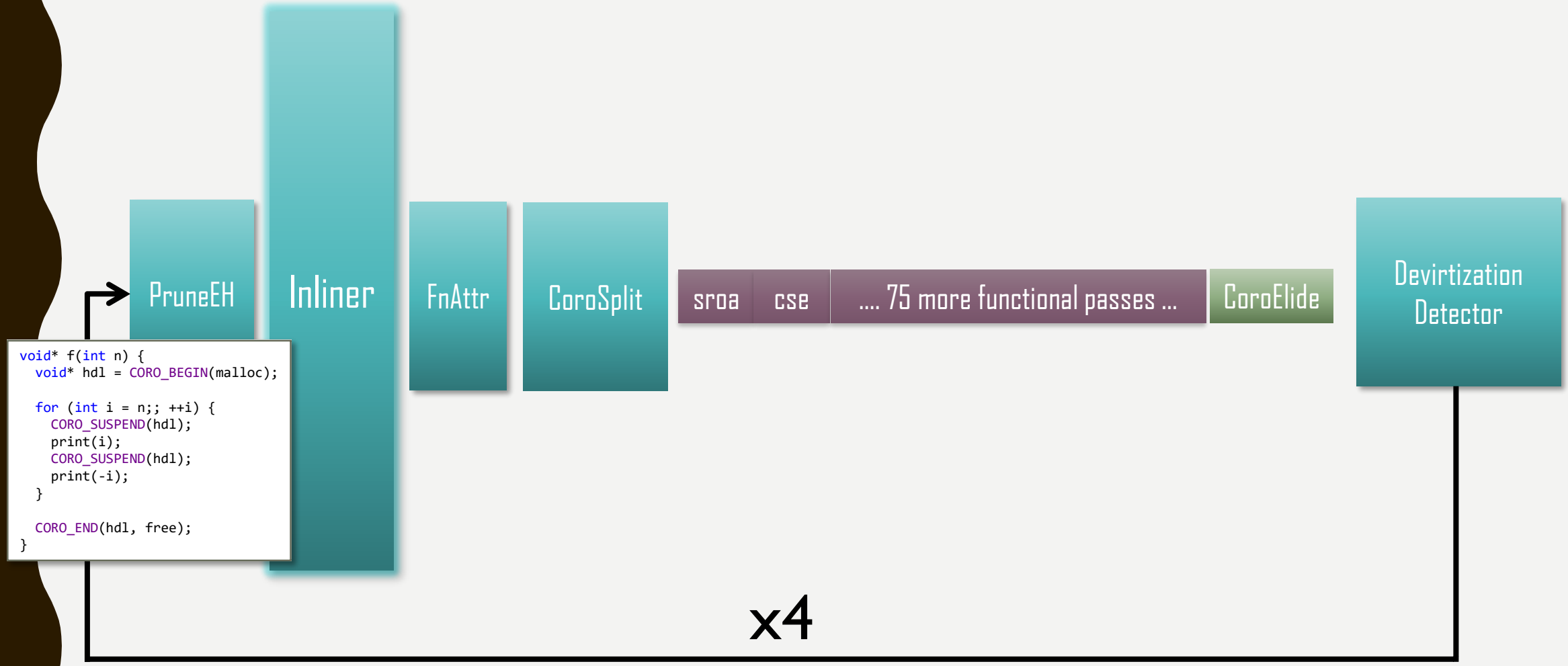
OPTIMIZING A SINGLE SCC



OPTIMIZING A SINGLE SCC

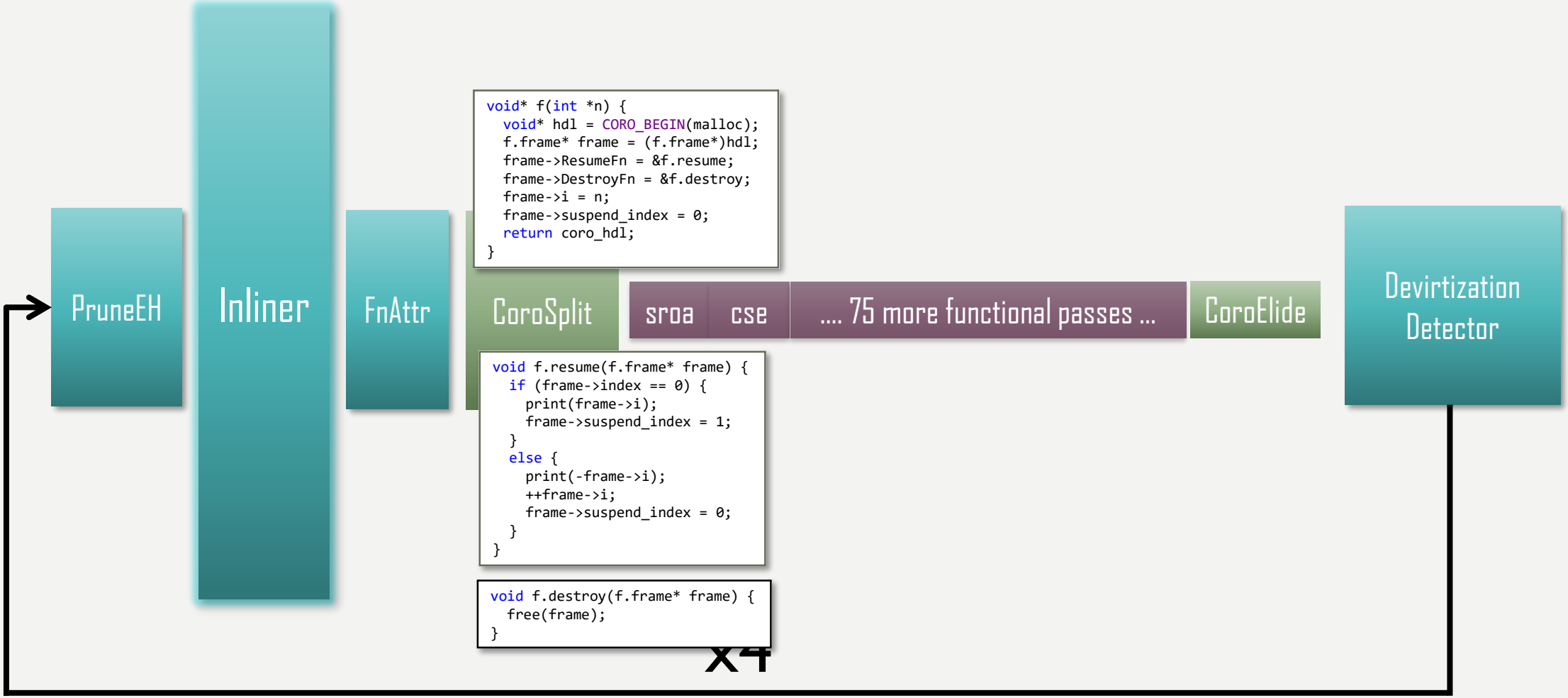


OPTIMIZING A SINGLE SCC




```
void* f(int n) {
  void* hdl = CORO_BEGIN(malloc);

  for (int i = n;; ++i) {
    CORO_SUSPEND(hdl);
    print(i);
    CORO_SUSPEND(hdl);
    print(-i);
  }

  CORO_END(hdl, free);
}
```

OPTIMIZING A SINGLE SCC

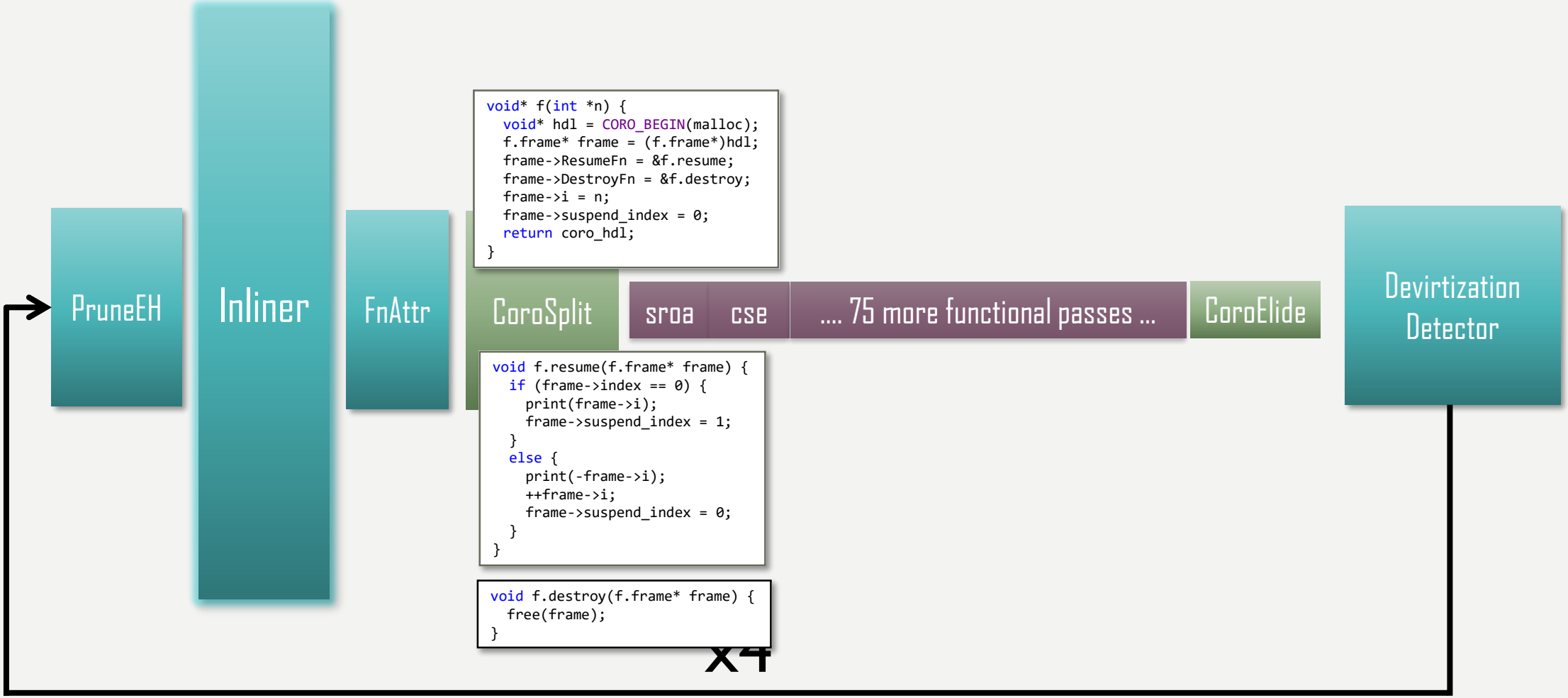



```
void* f(int *n) {
  void* hdl = CORO_BEGIN(malloc);
  f.frame* frame = (f.frame*)hdl;
  frame->ResumeFn = &f.resume;
  frame->DestroyFn = &f.destroy;
  frame->i = n;
  frame->suspend_index = 0;
  return coro_hdl;
}
```

```
void f.resume(f.frame* frame) {
  if (frame->index == 0) {
    print(frame->i);
    frame->suspend_index = 1;
  }
  else {
    print(-frame->i);
    ++frame->i;
    frame->suspend_index = 0;
  }
}
```

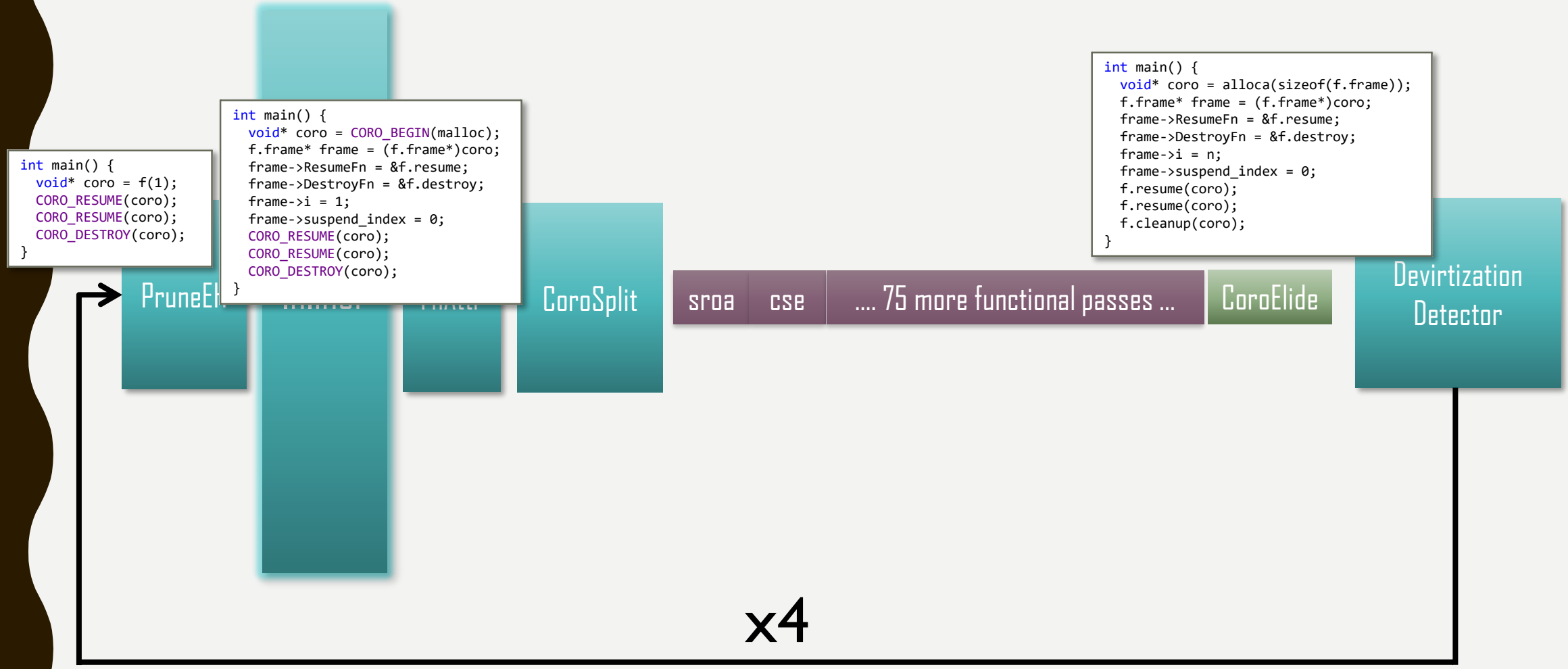
```
void f.destroy(f.frame* frame) {
  free(frame);
}
```

OPTIMIZING A SINGLE SCC



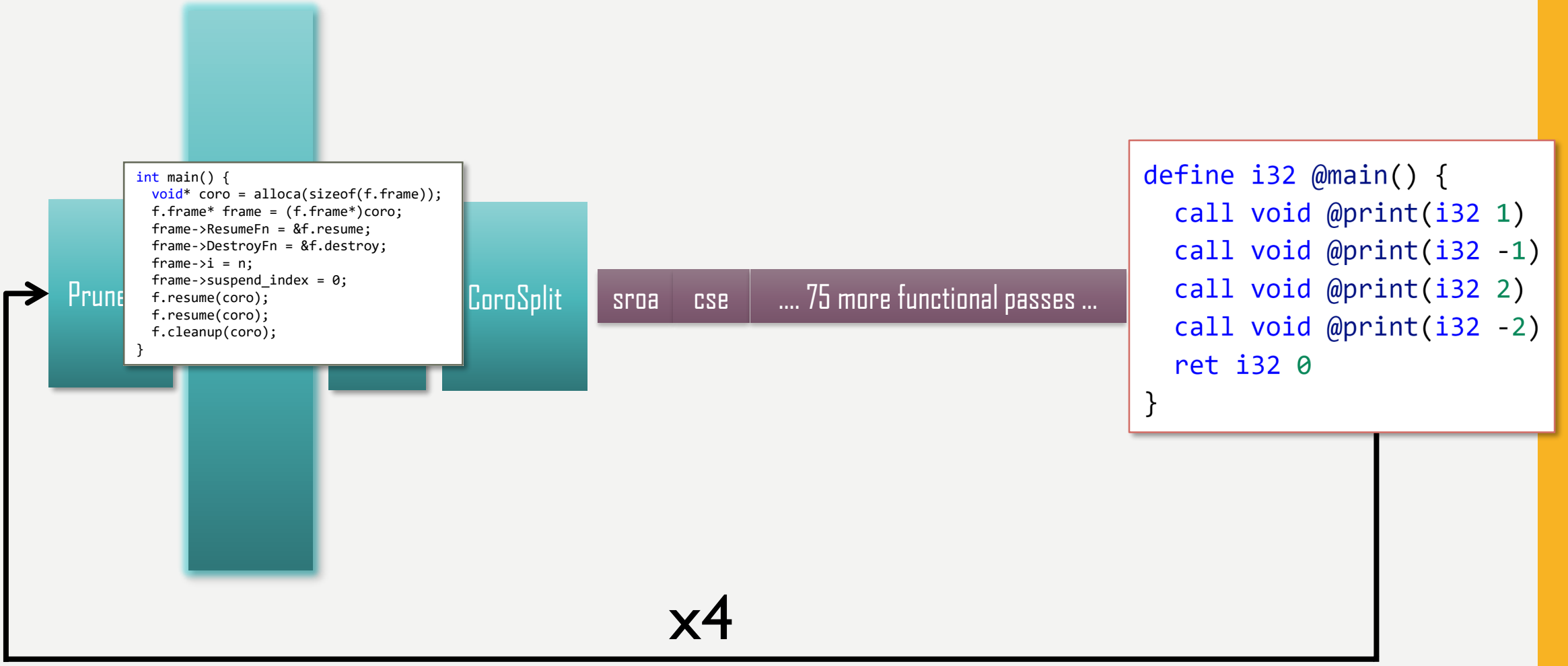
OPTIMIZING A SINGLE SCC

NOW LOOKING AT THE CALLER



OPTIMIZING A SINGLE SCC

NOW LOOKING AT THE CALLER



A NON-COROUTINE THAT CALLS A COROUTINE THAT CALLS A COROUTINE THAT CALLS A COROUTINE …

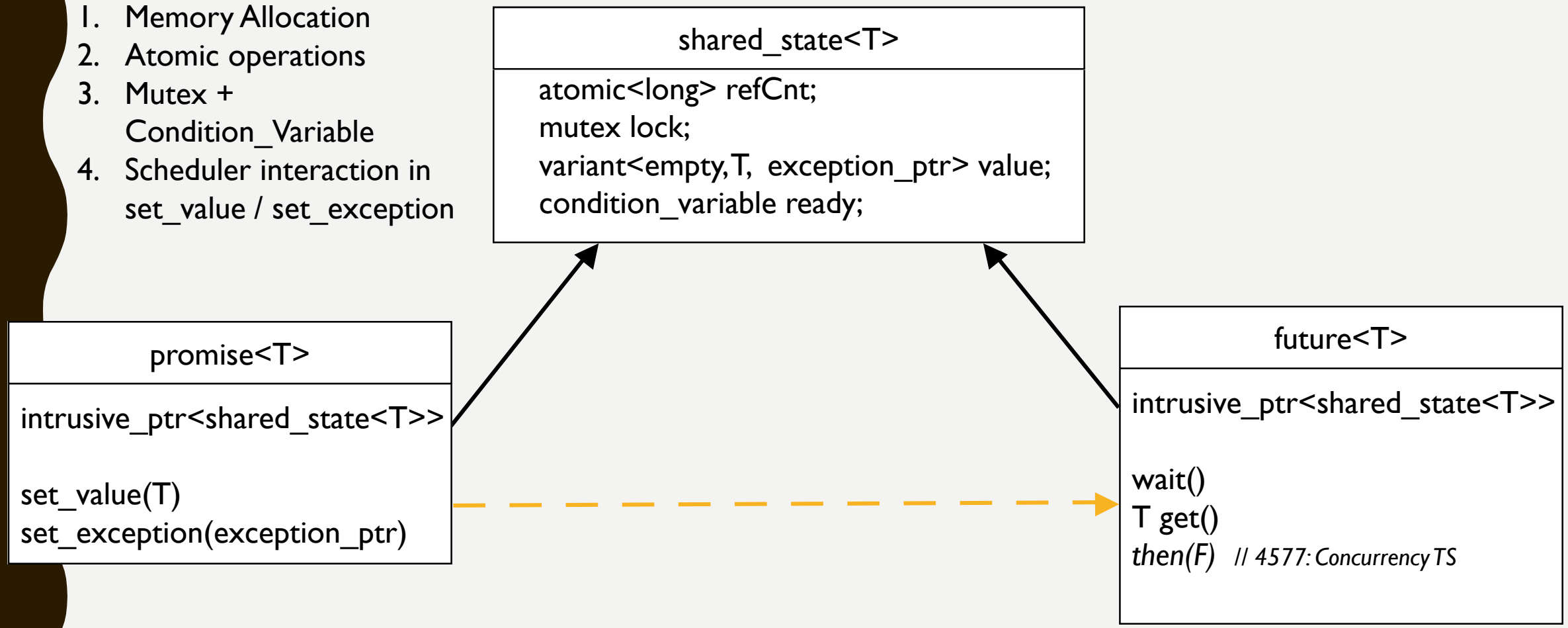
```
task<int> read_some(int n, char* buf);

task<void> read(int n, char* buf) {
  while (n > 0) {
    int read = co_await read_some(n, buf);
    buf += read;
    n -= read;
  }
}

task<void> do_work() {
  ...
  int read = co_await read(n, buf);
  ...
}
```

```
int main() {
  auto result = sync_await(do_work());
  printf("%d", result);
}
```

BUILDING A BETTER FUTURE

STD::FUTURE<T> AND STD::PROMISE<T>

1. Memory Allocation
2. Atomic operations
3. Mutex + Condition_Variable
4. Scheduler interaction in set_value / set_exception



REMINDER: C++ COROUTINE SUGAR

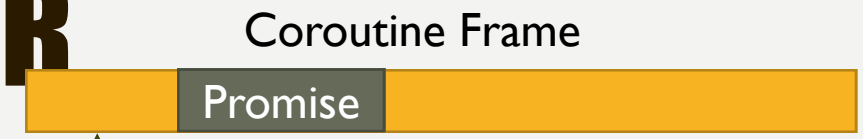
```
R f(Params) { body }

      using P = typename coroutine_traits<R, Params>::promise_type;

R f(Params) {
  P p;
  auto gro = p.get_return_object(); // returned when f returns to the caller
  co_await p.initial_suspend();
  body
final_suspend:
  co_await p.final_suspend();
}
```

REMINDER: C++ COROUTINE SUGAR

```
R f(Params) { body }

      using P = typename coroutine_traits<R, Params>::promise_type;

R f(Params) {
  P p;
  auto gro = p.get_return_object(); // returned when f returns to the caller
  co_await p.initial_suspend();
  try { body } catch (...) { p.set_exception(std::current_exception()); }
final_suspend:
  co_await p.final_suspend();
}
```

```
    co_return expr;   ➡   p.return_value(expr); goto final_suspend;
    co_return;        ➡   p.return_void(expr); goto final_suspend;
    co_yield expr;    ➡   co_await p.yield_value(expr);
```

REMINDER: MORE SUGAR




```
template <typename Promise = void>
struct coroutine_handle;

template <> struct coroutine_handle<void> {
  void *address() const;
  static coroutine_handle from_address(void *);
  void resume();
  void destroy();
  …
private:
  void *ptr;
};

template <typename Promise>
struct coroutine_handle<Promise>: coroutine_handle<> {
  Promise &promise() const;
  static coroutine_handle from_promise(Promise &);
  …
};
```

REMINDER: EVEN MORE SUGAR

```
auto r =
  co_await expr;
```

```
auto r =
 {
   auto && tmp = expr;
   if (!tmp.await_ready()) {
     __builtin_coro_save() // frame->suspend_index = n;
     tmp.await_suspend(<coroutine_handle>);
     __builtin_coro_suspend() // jmp epilog
   }
 resume_label_n:
   tmp.await_resume();
 };
```

```
struct suspend_always {
  bool await_ready() { return false; }
  void await_suspend(coroutine_handle<>) {}
  void await_resume() {}
};
```

```
struct suspend_never {
  bool await_ready() { return true; }
  void await_suspend(coroutine_handle<>) {}
  void await_resume() {}
};
```

WANT HEAP ELISION => RAII

```
template <typename T> struct task {
  struct promise_type {
    task get_return_object() { return {this}; }
    suspend_never initial_suspend() { return {}; }
    suspend_always final_suspend() { return {}; }
    template <typename U> void return_value(U &&value) {}
  };
  ~task() { coro.destroy(); }

private:
  task(promise_type *p)
      : coro(coroutine_handle<promise_type>::from_promise(*p)) {}

  coroutine_handle<promise_type> coro;
};
```

TAKE CARE OF RETURN VALUE

1. ~~Memory Allocation~~
2. ~~Atomic operations~~
3. Mutex + Conditional_Variable
4. Scheduler interaction in set_value / set_exception

```
template <typename T> struct task {
  struct promise_type {
    variant<monostate, T> result;

    task get_return_object() { return {this}; }
    suspend_never initial_suspend() { return {}; }
    suspend_always final_suspend() { return {}; }
    template <typename U> void return_value(U &&value) {
      result.template emplace<1>(std::forward<U>(value));
    }
  };

  T await_resume() { return get<1>(coro.promise().result); }
  … [ctor, dtor] …
  coroutine_handle<promise_type> coro;
};
```

ATTACH CONTINUATION

1. ~~Memory Allocation~~
2. ~~Atomic operations~~
3. ~~Mutex + Conditional_Variable~~
4. Scheduler interaction in set_value / set_exception

```
template <typename T> struct task {
  struct promise_type {
    variant<monostate, T> result;
    coroutine_handle<> waiter;
    task get_return_object() { return {this}; }
    suspend_always initial_suspend() { return {}; }
    suspend_always final_suspend() { return {}; }
    template <typename U> void return_value(U &&value) {
      result.template emplace<1>(std::forward<U>(value));
    }
  };
  bool await_ready() { return false; }
  void await_suspend(coroutine_handle<> CallerCoro) {
    coro.promise().waiter = CallerCoro;
    coro.resume();
  }
  T await_resume() { return get<1>(coro.promise().result); }
  … [ctor, dtor] …
  coroutine_handle<promise_type> coro;
};
```

TWEAK FINAL SUSPEND

1. ~~Memory Allocation~~
2. ~~Atomic operations~~
3. ~~Mutex + Conditional_Variable~~
4. ~~Scheduler interaction in set_value / set_exception~~

```
template <typename T> struct task {
  struct promise_type {
    variant<monostate, T> result;
    coroutine_handle<> waiter;
      …
    auto final_suspend() {
      struct Awaiter {
        promise_type* me;
        bool await_ready() { return false; }
        void await_suspend(coroutine_handle<>) {
           me->waiter.resume();
        }
        void await_resume() {}
      };
      return Awaiter{this};
    }
    template <typename U> void return_value(U &&value) {
      result.template emplace<1>(std::forward<U>(value));
    }
  };
```

```
tmp.await_suspend(<coroutine_handle>);
__builtin_coro_suspend() // jmp epilog
```

Tail Call

ADD EXCEPTION HANDLING

```
template <typename T> struct task {
  struct promise_type {
    variant<monostate, T> result;
    coroutine_handle<> waiter;
      …
    template <typename U> void return_value(U &&value) {
      result.template emplace<1>(std::forward<U>(value));
    }



  };
  T await_resume() {


    return get<1>(coro.promise().result);
  }
    …
  coroutine_handle<promise_type> coro;
}
```

ADD EXCEPTION HANDLING

```
template <typename T> struct task {
  struct promise_type {
    variant<monostate, T, exception_ptr> result;
    coroutine_handle<> waiter;
      …
    template <typename U> void return_value(U &&value) {
      result.template emplace<1>(std::forward<U>(value));
    }
    void set_exception(exception_ptr eptr) {
      result.template emplace<2>(std::move(eptr));
    }
  };
  T await_resume() {
    if (coro.promise().result.index() == 2)
      std::rethrow_exception(get<2>(coro.promise().result));
    return get<1>(coro.promise().result);
  }
    …
  coroutine_handle<promise_type> coro;
}
```

DONE

LLVM/CLANG COROUTINES GREAT THANKS TO:

Chandler Carruth

David Majnemer

Eli Friedman

Hal Finkel

Jim Radigan

Lewis Baker

Mehdi Amini

Richard Smith

Sanjoy Das

Victor Tong

MORE INFO

- LLVM Coroutines:
  http://llvm.org/docs/Coroutines.html
  experimental implementation is in the trunk of LLVM4.0

  opt flag –enable-coroutines to try them out
  Example: https://github.com/llvm-mirror/llvm/tree/master/test/Transforms/Coroutines

- C++ Coroutines:
  - http://wg21.link/P0057
  - MSVC – now
  - Clang Coroutines, soon, Clang 4.0 - possible

- (Not coroutine related but you can win XBox One S)
  Take this survey: http://aka.ms/cppcon

QUESTIONS?