The Present and Future of Interprocedural Optimization in LLVM

Stefanos Baziotis
stefanos.baziotis@gmail.com

Kuter Dinel
kuterdinel@gmail.com

Shinji Okumura
okuraofvegetable@gmail.com

Luofan Chen
clfbbn@gmail.com

Hideto Ueno
uenoku.tokotoko@gmail.com

Johannes Doerfert
johannesdoerfert@gmail.com

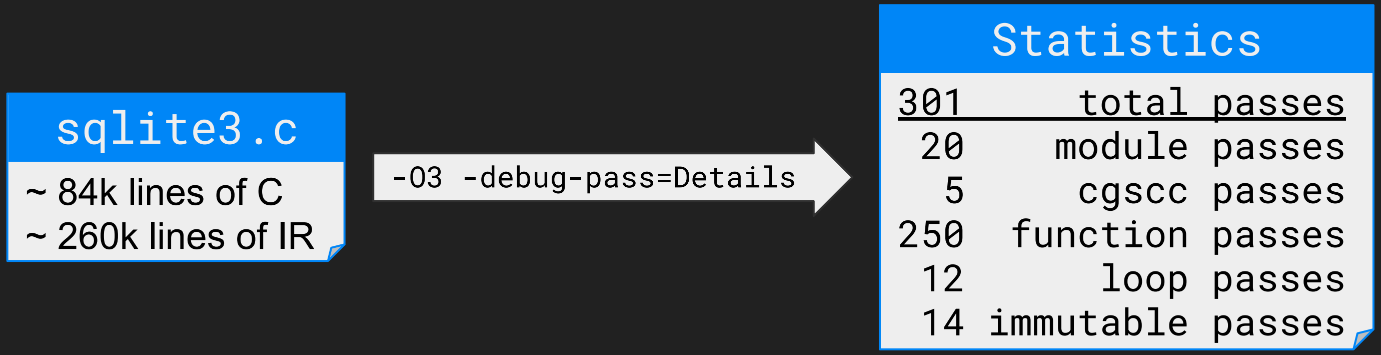
The Present

Kinds of IPO passes

- Inliner
  - AlwaysInliner, Inliner, InlineAdvisor, ...
- Propagation between caller and callee
  - Attributor[1], IP-SCCP, InferFunctionAttrs, ArgumentPromotion, DeadArgumentElimination, ...
- Linkage and Globals
  - GlobalDCE, GlobalOpt, GlobalSplit, ConstantMerge, ...
- Others
  - MergeFunction, OpenMPOpt[2], HotColdSplitting[3], Devirtualization[4]...

Checkout the IPO tutorial[5] for details!

Current State of IPO in LLVM



sqlite3.c

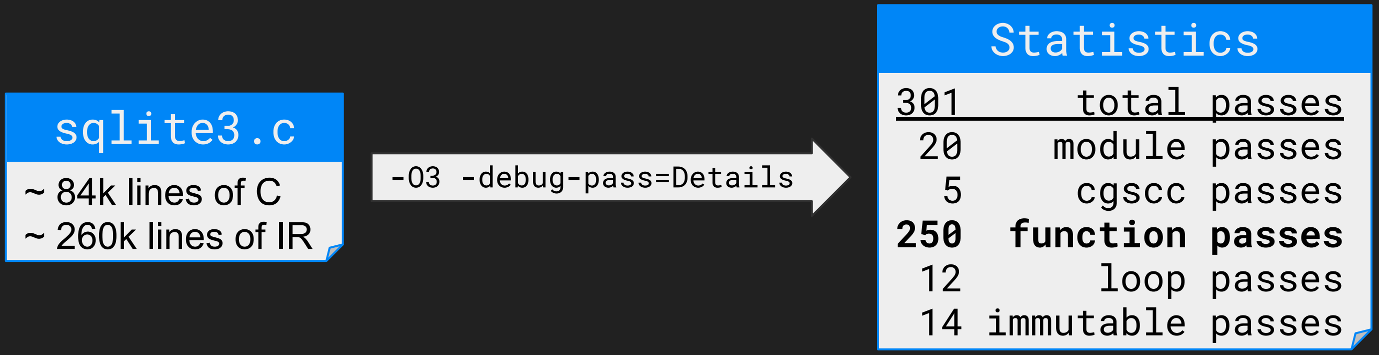
~ 84k lines of C
~ 260k lines of IR

```
-O3 -debug-pass=Details
```

Statistics

301	total passes
20	module passes
5	cgscc passes
250	function passes
12	loop passes
14	immutable passes

Current State of IPO in LLVM



sqlite3.c

~ 84k lines of C
~ 260k lines of IR

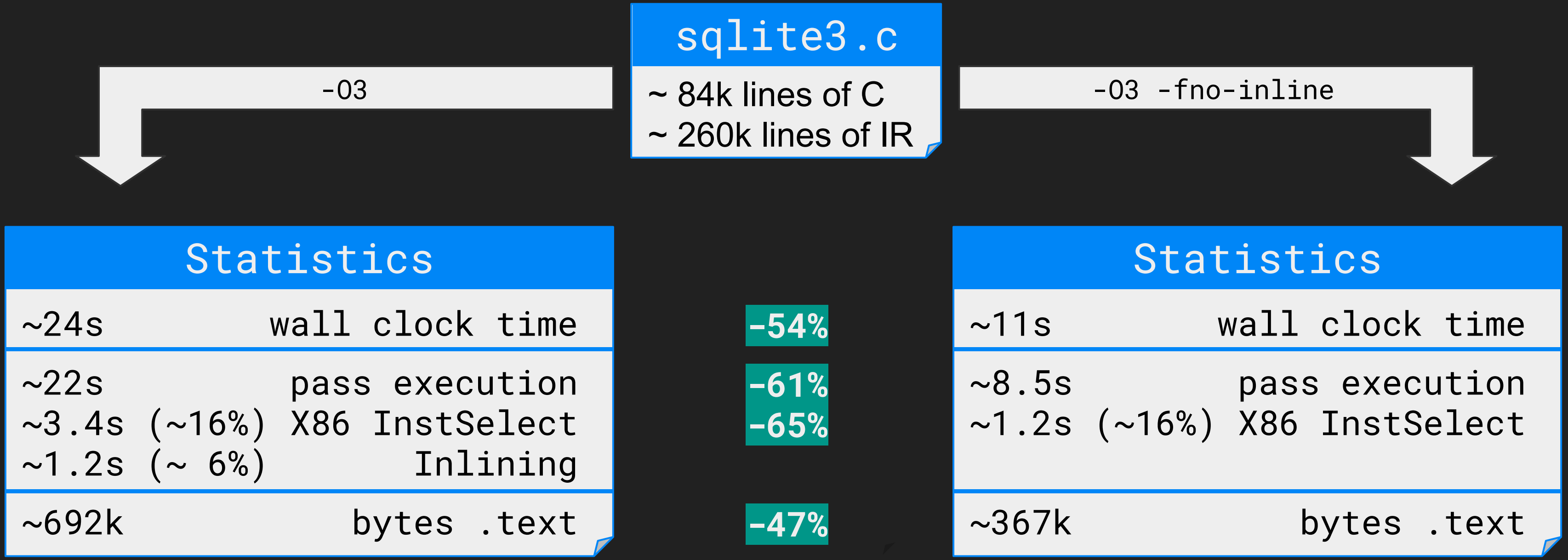
`-O3 -debug-pass=Details`

Statistics

301	total passes
20	module passes
5	cgscc passes
250	**function passes**
12	loop passes
14	immutable passes

>90% of passes are intraprocedural

Current State of IPO in LLVM



sqlite3.c

~ 84k lines of C
~ 260k lines of IR

-O3

-O3 -fno-inline

Statistics

~24s wall clock time

~22s pass execution
~3.4s (~16%) X86 InstSelect
~1.2s (~ 6%) Inlining

~692k bytes .text

-54%

-61%

-65%

-47%

Statistics

~11s wall clock time

~8.5s pass execution
~1.2s (~16%) X86 InstSelect

~367k bytes .text

>50% time & bytes spend as a *consequence* of inlining

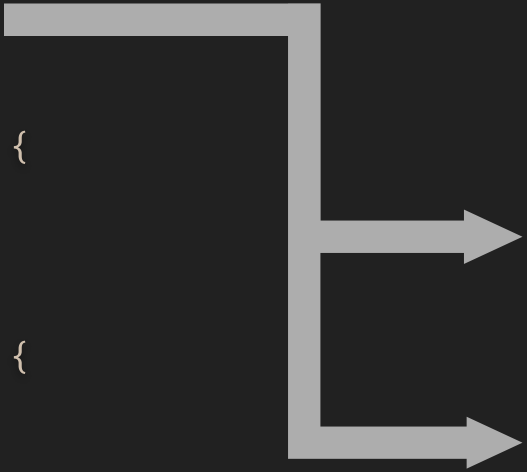
Inlining - Benefits: Code specialization

```
static void foo(int x, bool c) {
  if (c) y = 1; else y = 2;
  use(x, y);

}

void caller1(int x) {
   foo(x, true);

}

void caller2(int x) {
   foo(x, false);

}
```

```
void caller1(int x) {
  use(x, 1);

}

void caller2(int x) {
  use(x, 2);

}
```

Inlining - Drawbacks: Code Duplication

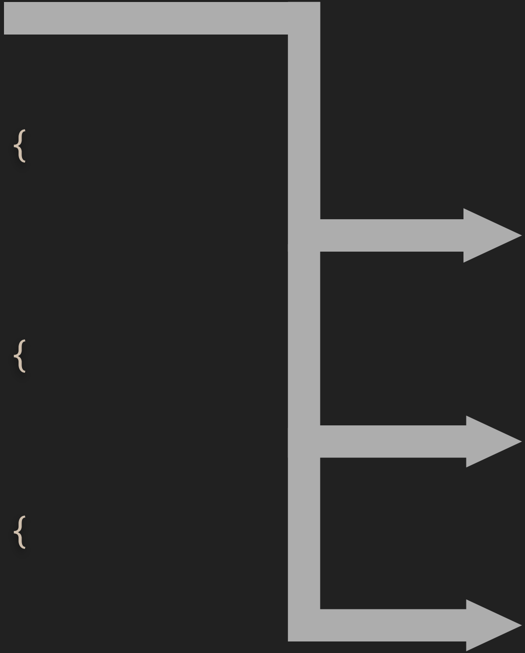
```
static void foo(int x, bool c) {
  if (c) y = 1; else y = 2;
  use(x, y);
  /* more stuff */
}

void caller1(int x) {
   foo(x, true);

}

void caller2(int x) {
   foo(x, false);

}
```



```
void caller1(int x) {
  use(x, 1);
  /* more stuff */
}

void caller2(int x) {
  use(x, 2);
  /* more stuff */
}
```

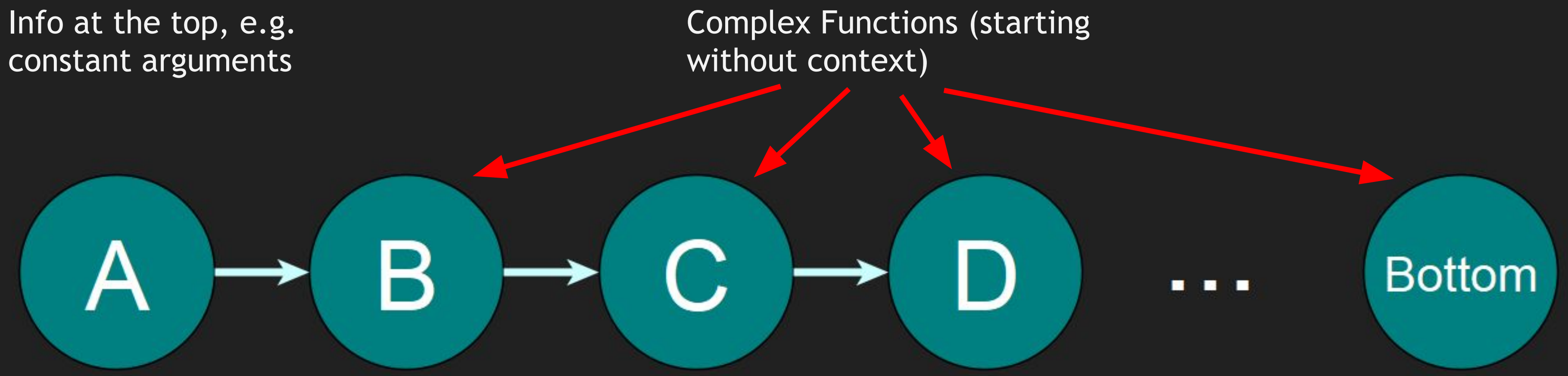
Inlining - Drawbacks: Code Duplication

```
static void foo(int x, bool c) {
  if (c) y = 1; else y = 2;
  use(x, y);
  /* more stuff */
}

void caller1(int x) {
   foo(x, true);

}

void caller2(int x) {
   foo(x, false);

}
void caller3(int x) {
   foo(x, false);

}
```



```
void caller1(int x) {
  use(x, 1);
  /* more stuff */
}

void caller2(int x) {
  use(x, 2);
  /* more stuff */
}
void caller3(int x) {
  use(x, 2);
  /* more stuff */
}
```

Inlining - Drawbacks: Inline Order

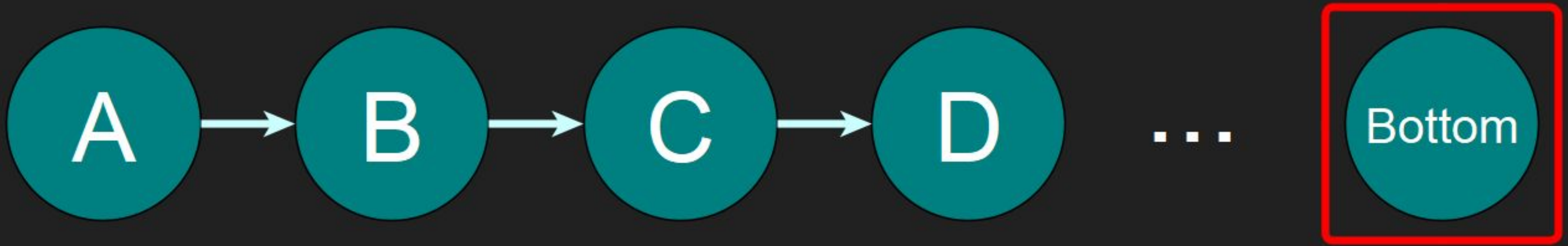


Info at the top, e.g. constant arguments

Complex Functions (starting without context)

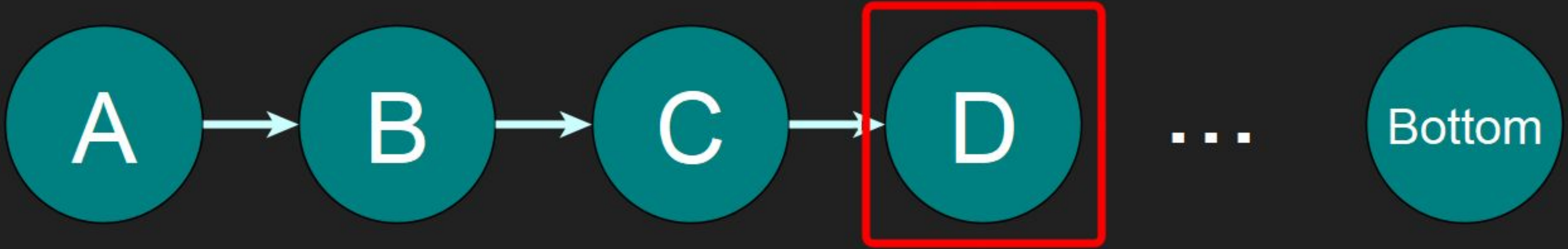
Inlining - Drawbacks: Inline Order

Info at the top, e.g.
constant arguments



Inlining - Drawbacks: Inline Order

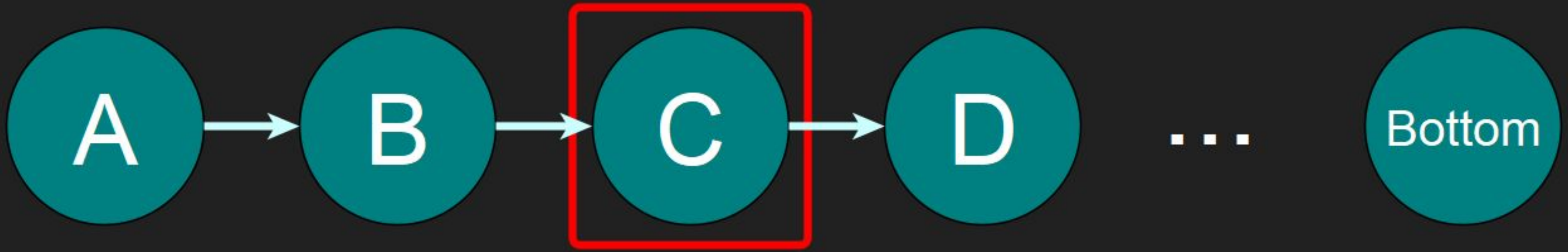
Info at the top, e.g.
constant arguments



Inlining - Drawbacks: Inline Order

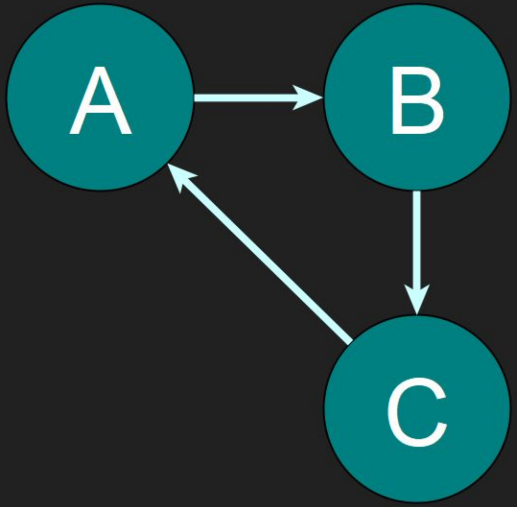
Info at the top, e.g.
constant arguments

Maybe the inliner
stops here

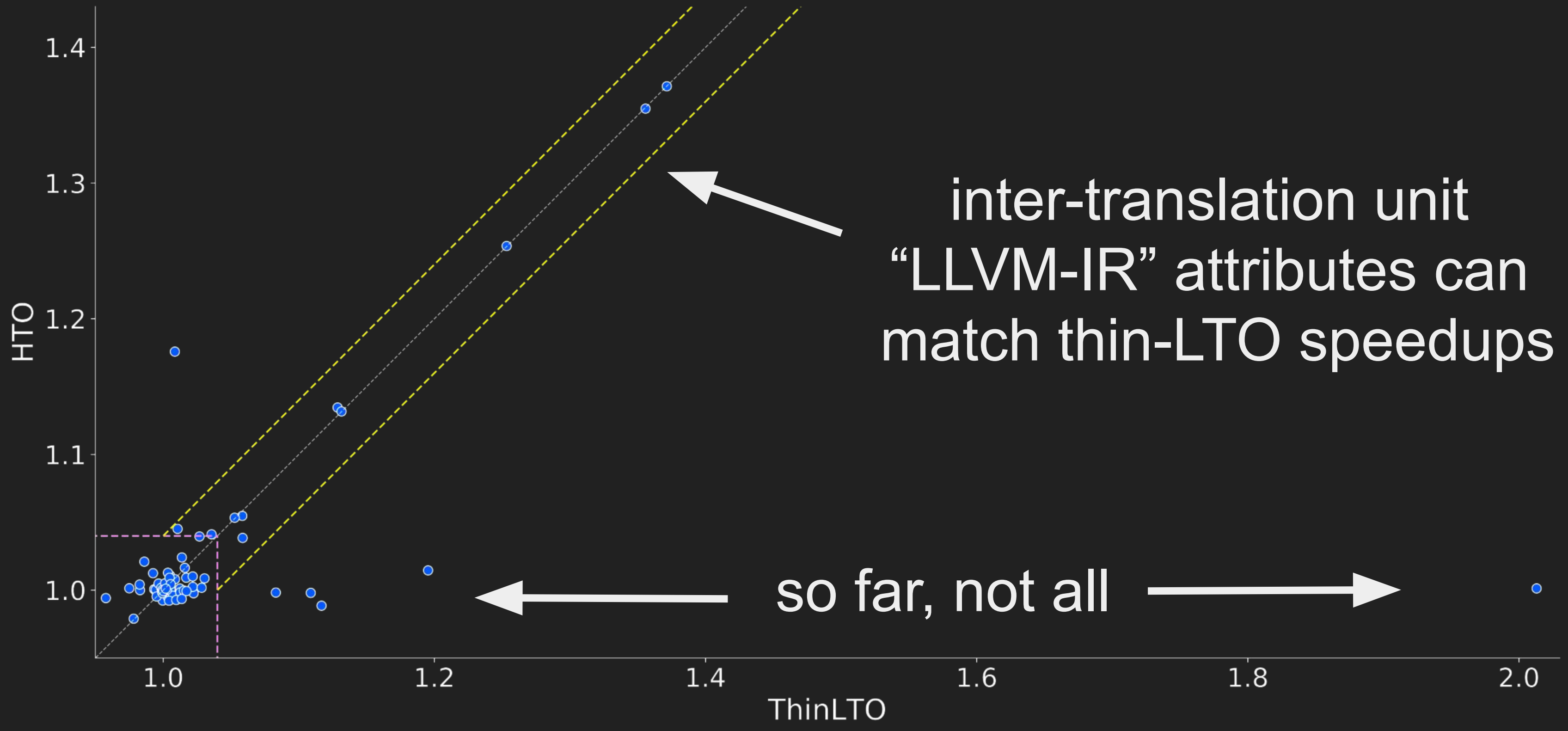


Inlining - Drawbacks: Inline Order

Strongly Connected Components (SCCs) have no top-down/bottom-up order



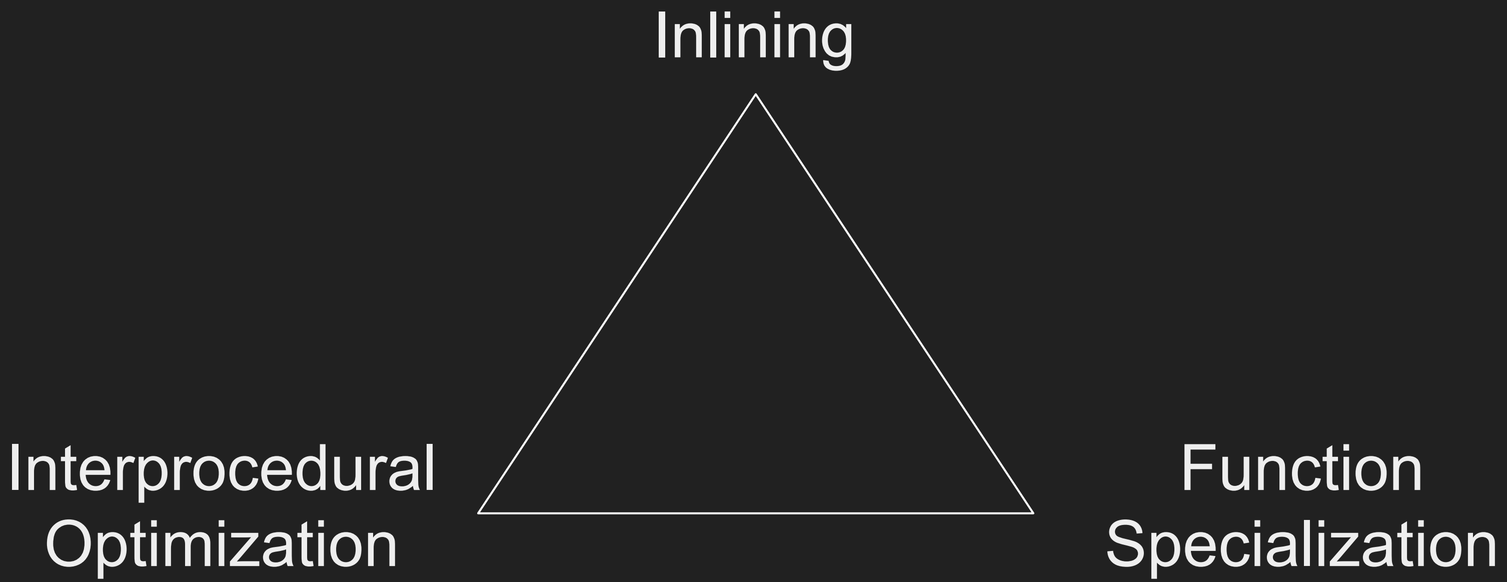
Inlining - Alternatives: thin-LTO[7] vs HTO[8]



inter-translation unit “LLVM-IR” attributes can match thin-LTO speedups

so far, not all

Design Space

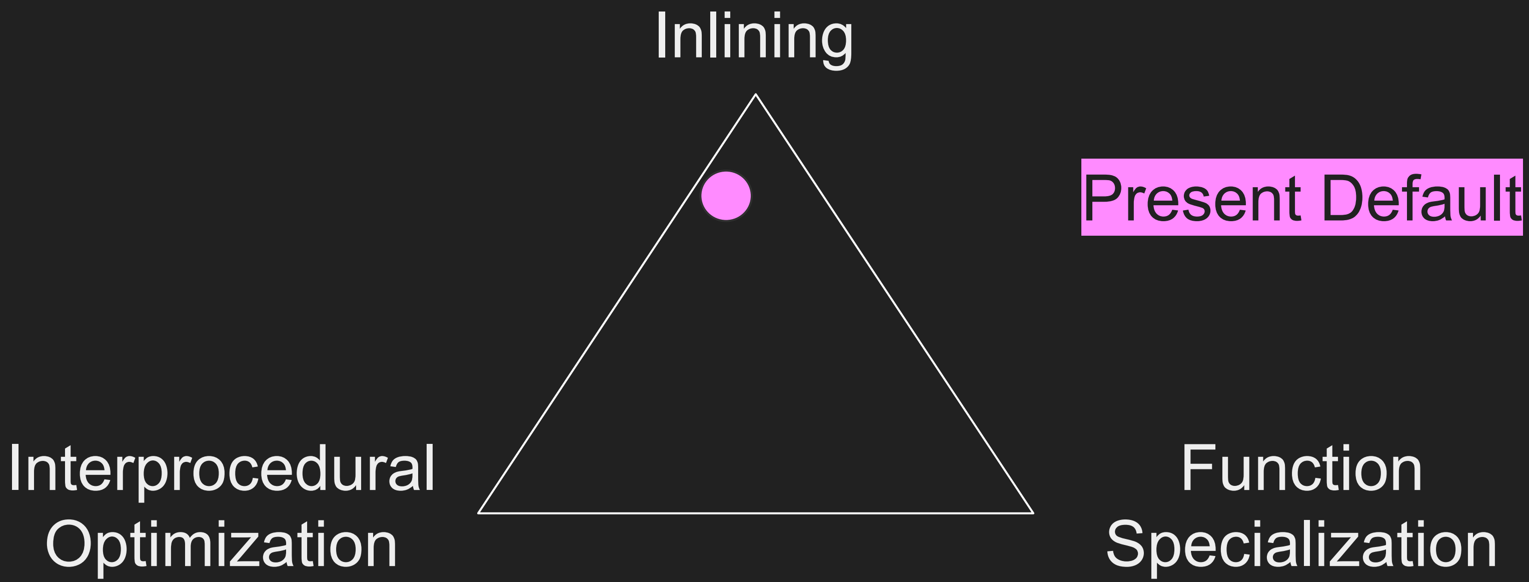


Inlining

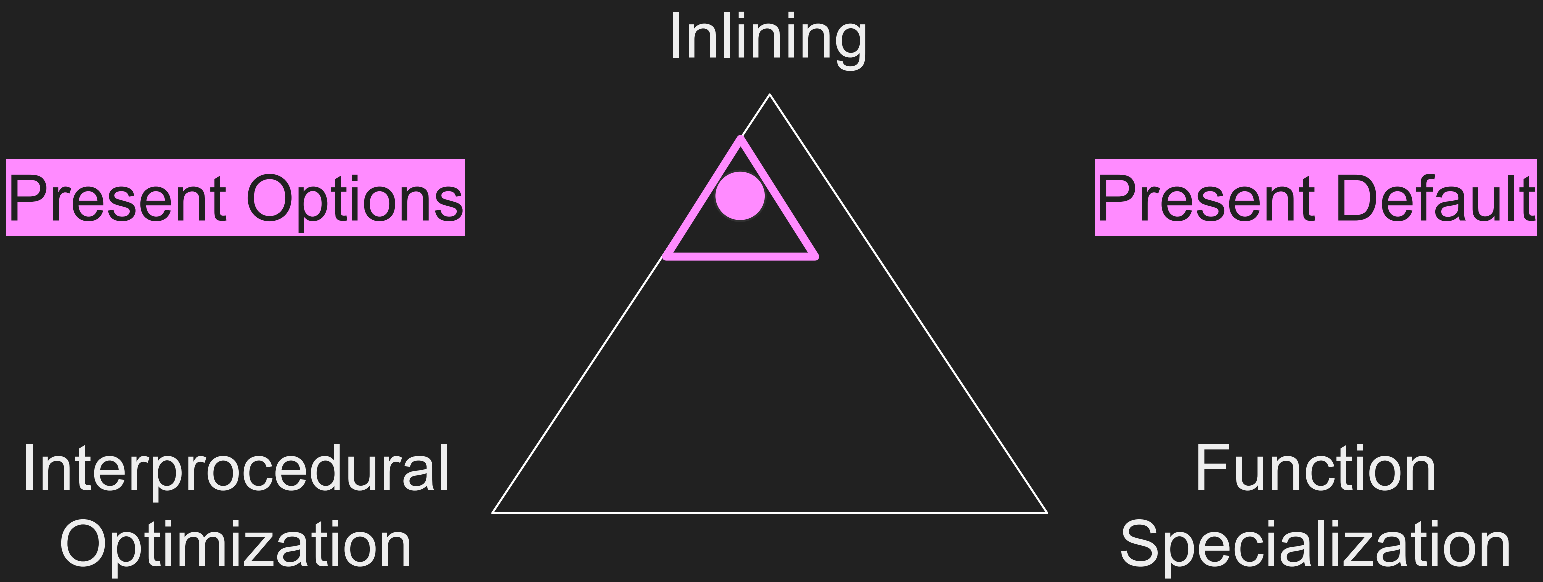
Interprocedural Optimization

Function Specialization

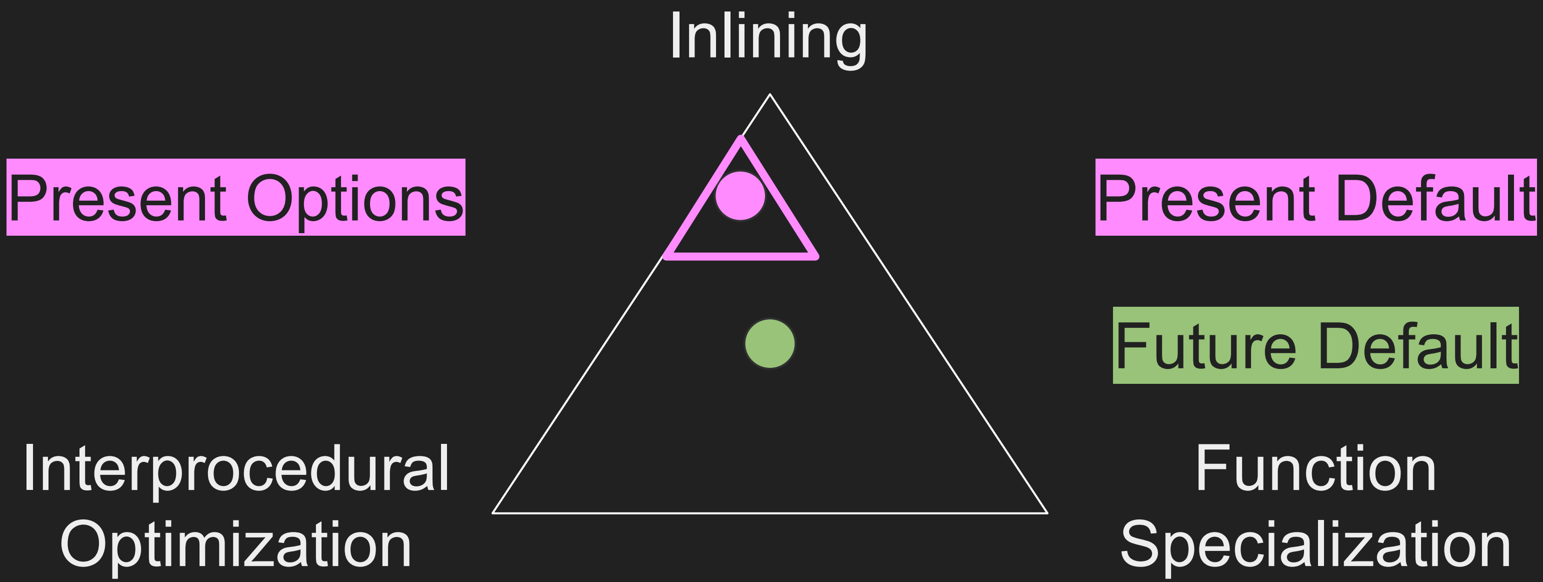
Design Space



Design Space



Design Space



Inlining

Present Options

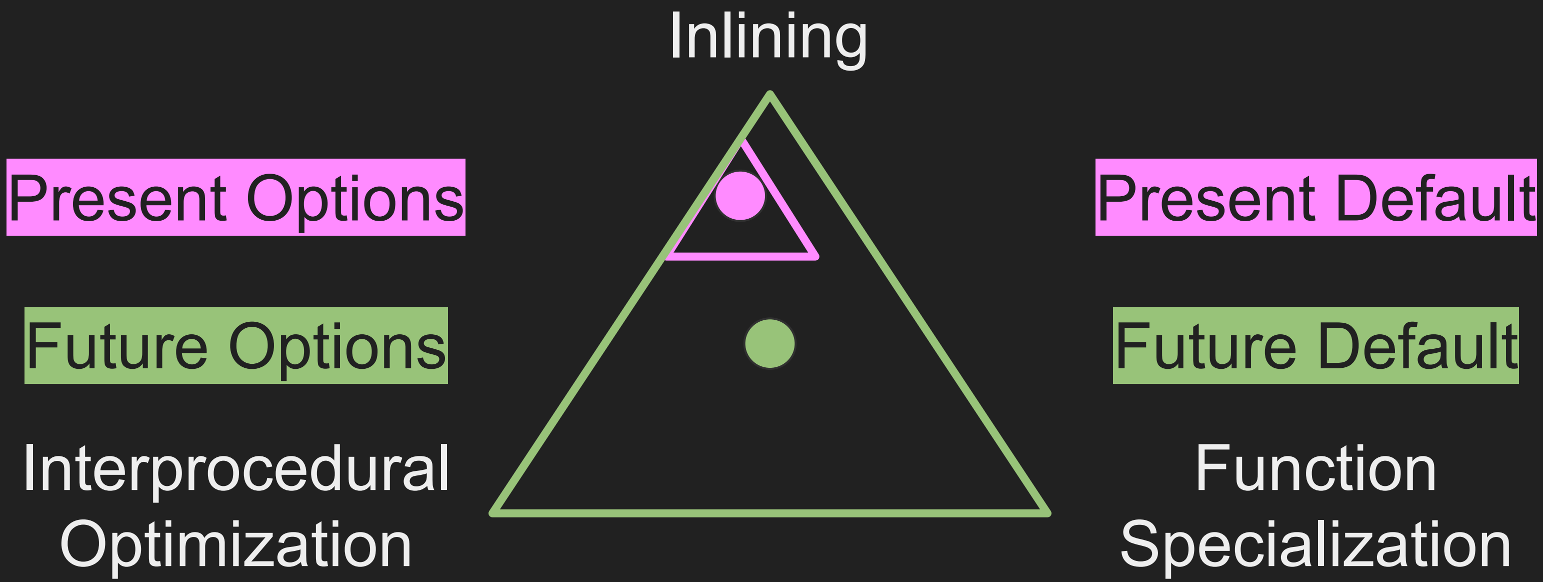
Present Default

Future Default

Interprocedural Optimization

Function Specialization

Design Space



Inlining

Present Options

Present Default

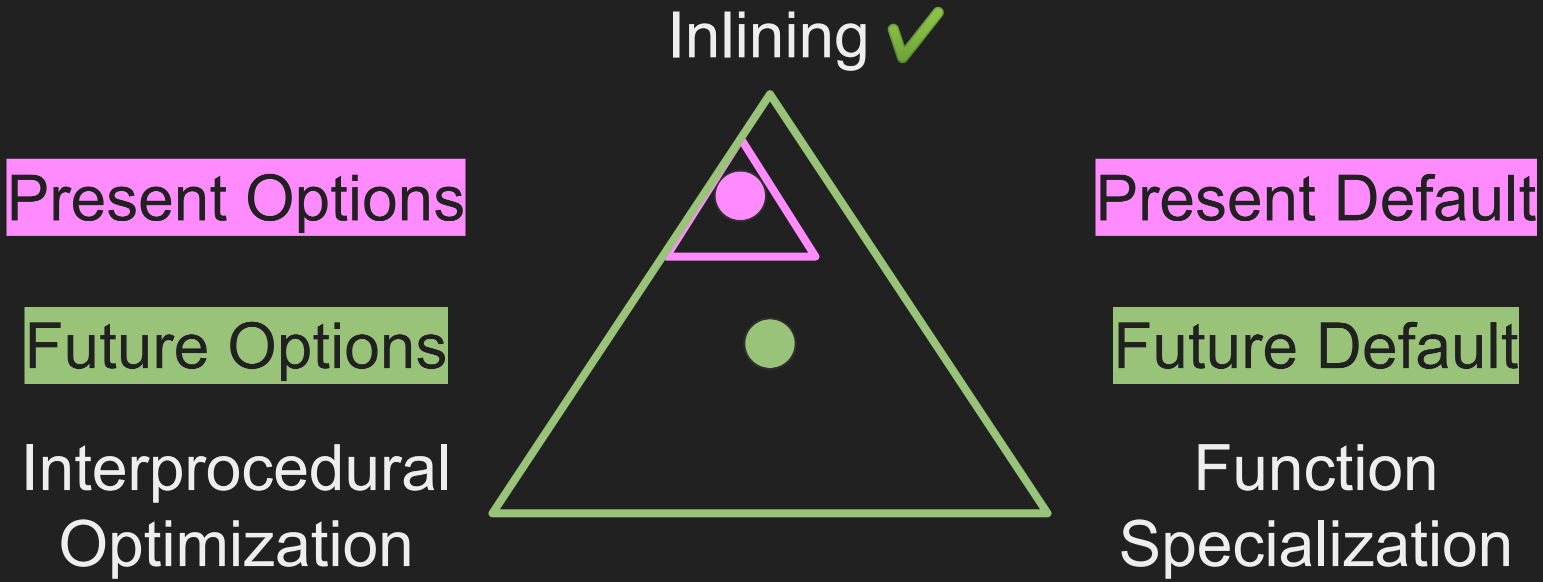
Future Options

Future Default

Interprocedural Optimization

Function Specialization

Design Space



Inlining ✓

Present Options

Present Default

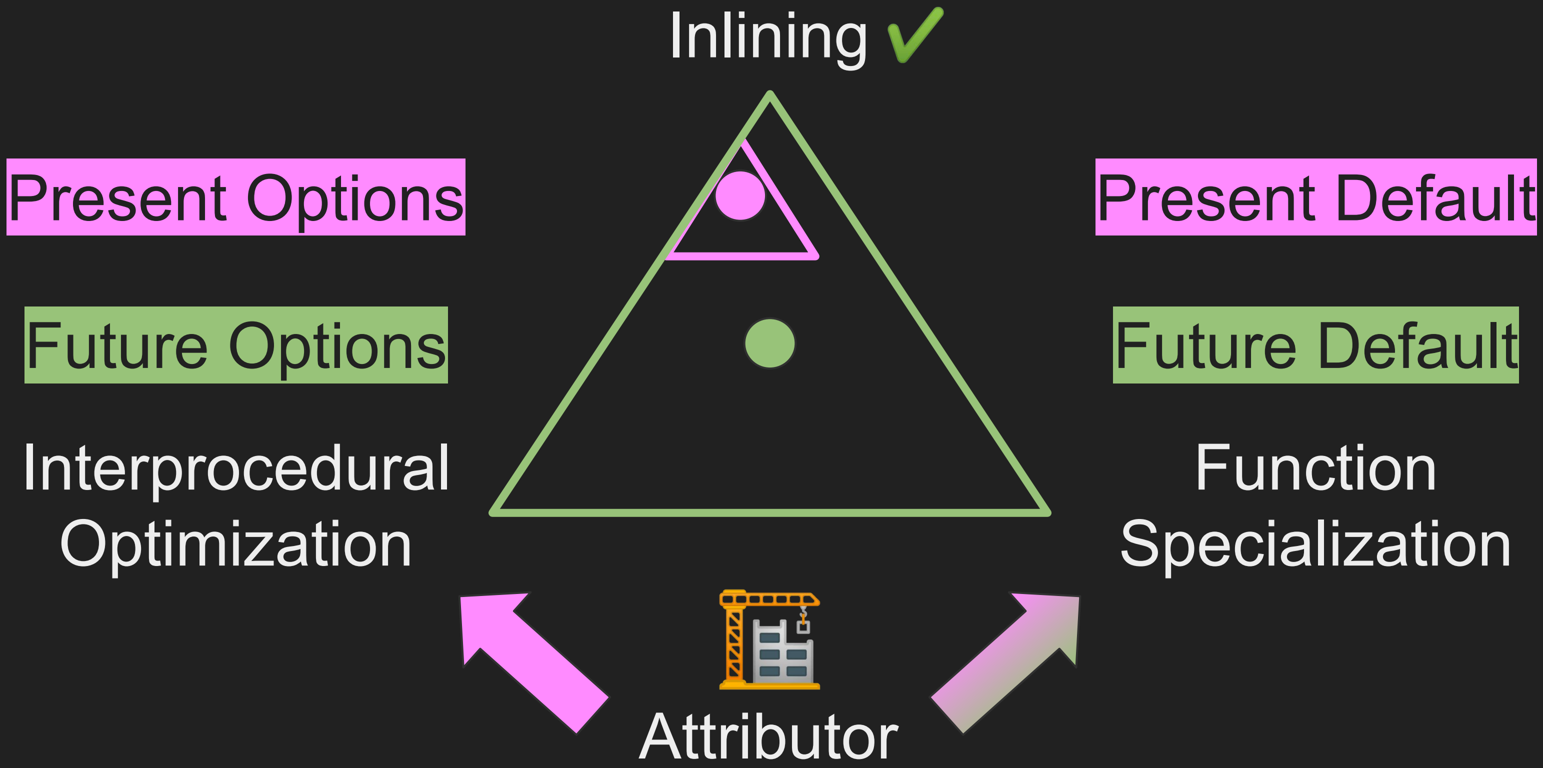
Future Options

Future Default

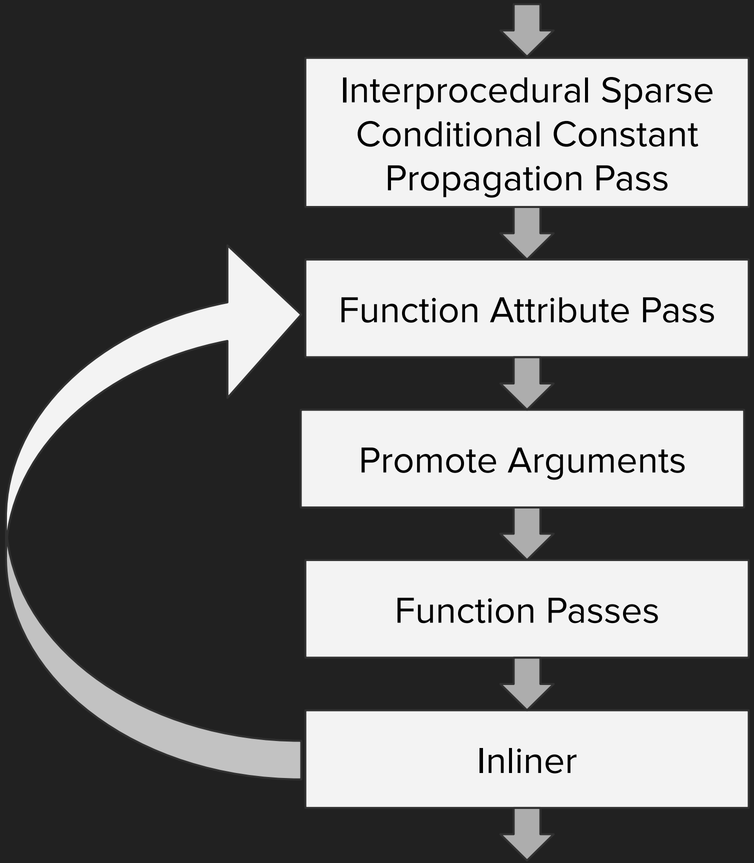
Interprocedural Optimization

Function Specialization

Design Space



Pass Ordering



Interprocedural Sparse Conditional Constant Propagation Pass

Function Attribute Pass

Promote Arguments

Function Passes

Inliner

```
void unknown(int &x);

static void check_n_rec(int n, int &x, int &y) {
  if (x) unknown(x);
  if (n) check_n_rec(n-1, y, x);
}

int test(int n) {
  int x = 0, y = 0;
  check_n_rec(n, x, y);
  return x + y;
}
```

The Future

Attributor

The Attributor[1,9] is an *interprocedural fixpoint iteration framework*; with lots of built-in features.

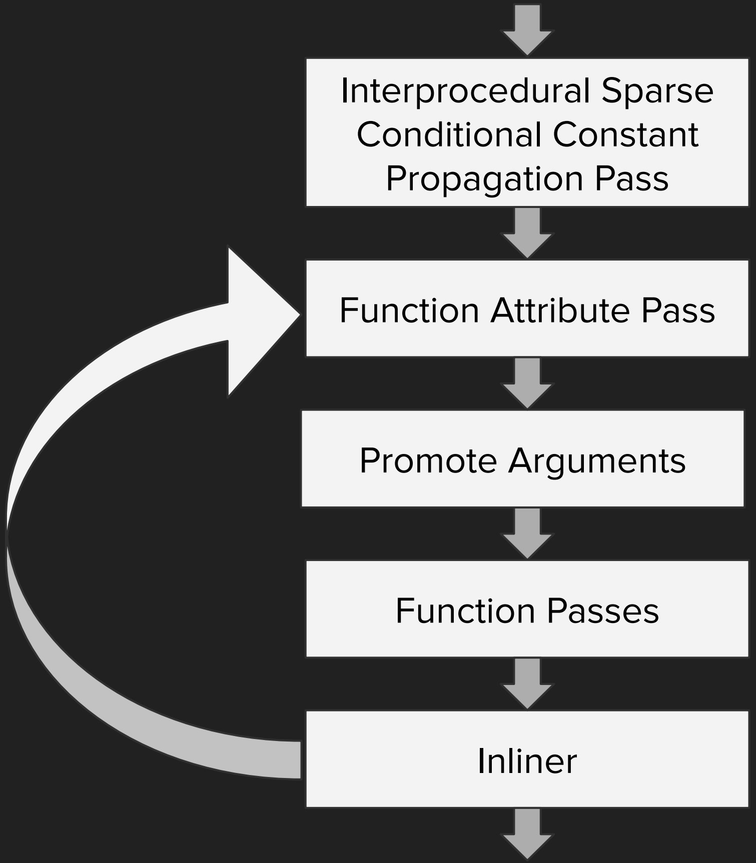
Attributor covers many IPO passes

- infers almost all LLVM-IR attributes
  ✓ (Reverse)Post Order Function Attribute Pass

- simplifies arguments, branches, return values and ...
  ✓ IP-SCCP*, Called Value Propagation

- rewrites function signatures
  ✓ Argument Promotion, Dead Argument Elimination

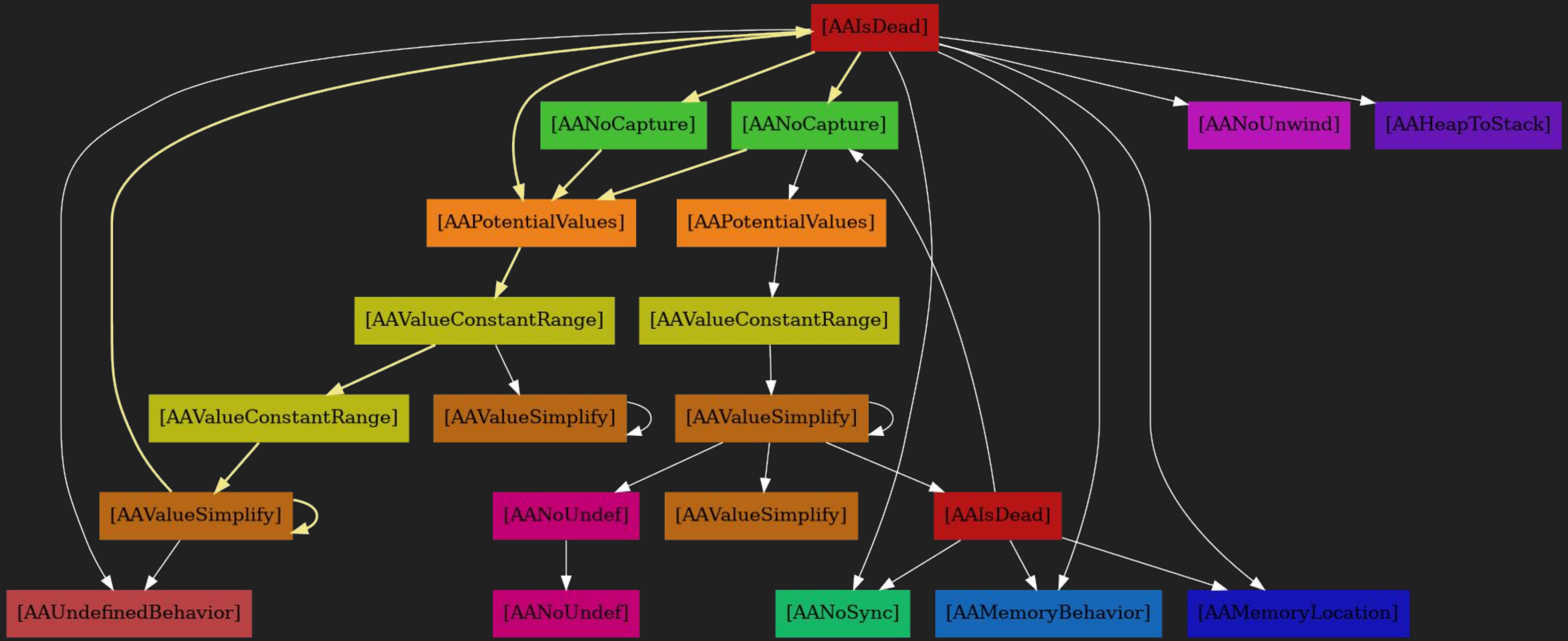
Pass Ordering



```
void unknown(int &x);

static void check_n_inc(int n, int &x, int &y) {
  if (x) unknown(x);
  if (n) check_n_inc(n-1, y, x);
}

int test(int n) {
  int x = 0, y = 0;
  check_n_inc(n, x, y);
  return x + y;
}
```

Dataflow Iterations

```
void unknown(int &x);
static void check_n_inc(int n, int &x, int &y) {
  if (x) unknown(x);
  if (n) check_n_inc(n-1, y, x);
}

int test(int n) {
  int x = 0, y = 0;
  check_n_inc(n, x, y);
  return x + y;
}
```



Function Specialization

```
__attribute__((linkonce_odr))
void foo(int x, bool c) {
  if (c) y = 1; else y = 2;
  use(x, y);
}



void caller1(int x) {
  foo(x, false);
}
void caller2(int x) {
  foo(x, false);
}
void caller3(int x) {
  foo(x, true);
}
```

```
__attribute__((linkonce_odr))
void foo(int x, bool c) {
  if (c) y = 1; else y = 2;
  use(x, y);
}
static void foo.internal(int x, bool c) {
  if (c) y = 1; else y = 2;
  use(x, y);
}

void caller1(int x) {
  foo.internal.false(x);
}
void caller2(int x) {
  foo.internal.false(x);
}
void caller3(int x) {
  foo.internal.true(x);
}
```

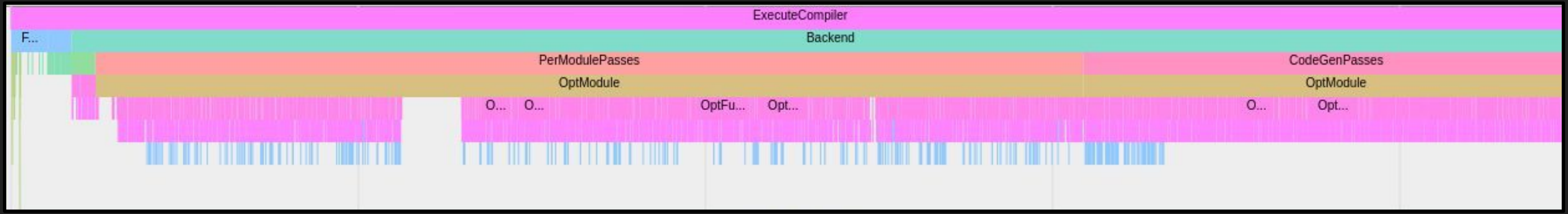
Function Specialization

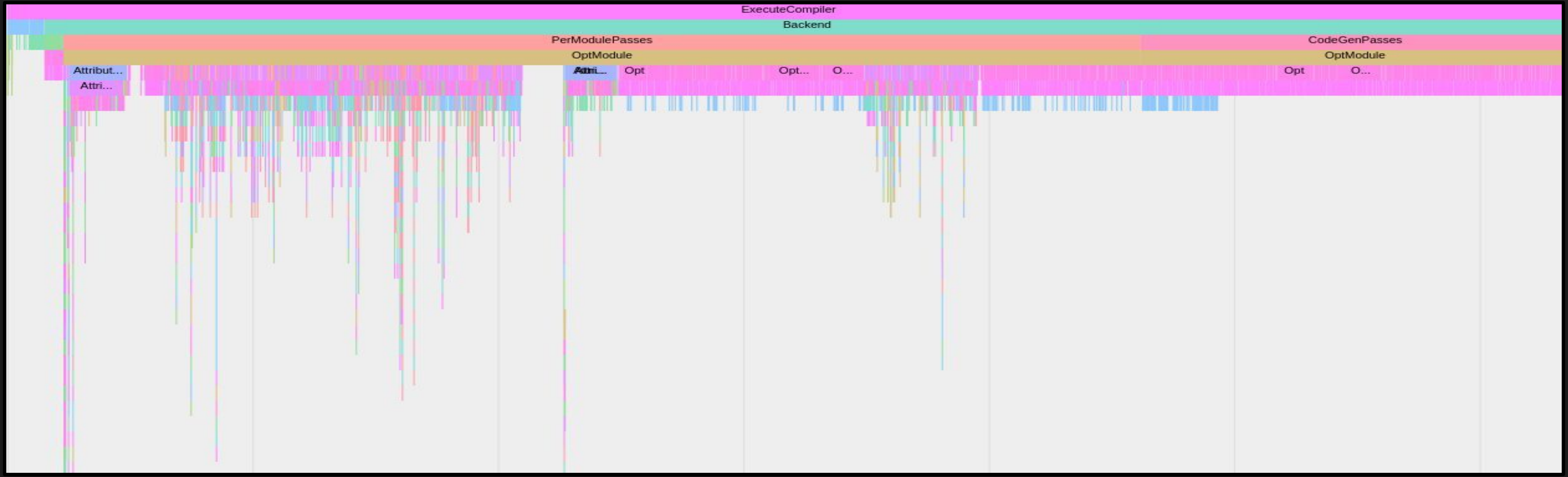
```
__attribute__((linkonce_odr))
void foo(int x, bool c) {
  if (c) y = 1; else y = 2;
  use(x, y);
}
```

```
void caller1(int x) {
  foo(x, false);
}
void caller2(int x) {
  foo(x, false);
}
void caller3(int x) {
  foo(x, true);
}
```

```
__attribute__((linkonce_odr))
void foo(int x, bool c) {
  if (c) y = 1; else y = 2;
  use(x, y);
}
static void foo.internal.false(int x) {
  use(x, 2);
}
static void foo.internal.true(int x) {
  use(x, 1);
}
void caller1(int x) {
  foo.internal.false(x);
}
void caller2(int x) {
  foo.internal.false(x);
}
void caller3(int x) {
  foo.internal.true(x);
}
```

Time Traces


ExecuteCompiler
F...
Backend
PerModulePasses
CodeGenPasses
OptModule
OptModule
O...
O...
OptFu...
Opt...
O...
Opt...


ExecuteCompiler
Backend
PerModulePasses
CodeGenPasses
OptModule
OptModule
Attribut...
Attri...
Attri...
Opt
Opt...
O...
Opt
O...

How To Get There

Intrinsic & Library Functions

State

- *Most* intrinsics & library functions have *some* attributes

Intrinsic & Library Functions

State

- ~~*Most* intrinsics & library functions have *some* attributes~~
- *Most* intrinsics & library functions miss *a lot* of attributes

Intrinsic & Library Functions

State

- ~~*Most* intrinsics & library functions have *some* attributes~~
- *Most* intrinsics & library functions miss *a lot* of attributes

Solutions (in progress)

- Default attributes for intrinsics, you need to opt-out
- Revisit library functions and add attributes systematically

Intrinsic & Library Functions

llvm-test-suite/SingleSource/Benchmarks/BenchmarkGame/fannkuch.c

```
[Heap2Stack] Bad user:  call void @llvm.memcpy.p0i8.p0i8.i64(...) may-free  the allocation
[Heap2Stack] Bad user:  call void @llvm.memcpy.p0i8.p0i8.i64(...) may-free  the allocation
[Heap2Stack]: Removing calloc call:  %call = call noalias dereferenceable_or_null(44)
                                                 i8* @calloc(i64 noundef 11, i64 noundef 4)
```

3x heap to stack + follow up transformations:
~5% speedup

Introduce & Utilize New Attributes

Frontend:

- generic LLVM-IR attributes[8]
- “access” (like GCC[10])

Introduce & Utilize New Attributes

Frontend:

- generic LLVM-IR attributes[8], i.a., `__attribute__((fn_arg(“willreturn”)))`
- “access” (like GCC[10]), i.a., `__attribute__ ((access (read_only, 1))) int puts (const char*)`

Introduce & Utilize New Attributes

Frontend:

- generic LLVM-IR attributes[8], i.a., `__attribute__((fn_arg("willreturn")))`
- "access" (like GCC[10]), i.a., `__attribute__ ((access (read_only, 1))) int puts (const char*)`

LLVM-IR:

- fine-grained memory effects:
  - `writes(@errno,...)`
  - `2^{inaccessible,argument,global,...}`
- `potential values`
  - `value(null, arg(0), @global, ...)`

Attributor - Testing

State

- *reasonable* unit test coverage
- *no* regular (=CI) builds

Solutions

- Try it out, report and track down bugs
- Setup buildbot(s) that enable the Attributor (anyone?)

Attributor - Memory Overhead

State

- *Way* better than in the last release
- Mostly an issue for the module-wide pass, not the call graph pass

Solutions (in progress)

- Drop Attributor state that is not useful anymore eagerly
- Minimize the number of Abstract Attributes created

Attributor - Compile Time Overhead

State

- *Improved* compared to the last release
- Issue for both the module-wide pass and the call graph pass

Solutions (in progress)

- Improve the schedule order (less updates, better locality, ...)
- Avoid costly deductions or perform them conditionally
- Minimize the number of Abstract Attributes created

Attributor - Selective Investment

Focus on hot code; look at otherwise cold code only as a consequence

Attributor - Selective Investment

Focus on hot code; look at otherwise cold code only as a consequence

```c
static void foo() { ... }
static int* bar() { ...; return ...; }
static void baz(int *) { ... }

extern void __attribute__((cold)) sink();
void hotcold(int cond) {
  int *p = ...;
  if (cond) {
    p = bar();
    sink();
    foo();
  }
  baz(p);
}
```

Attributor - Selective Investment

Focus on hot code; look at otherwise cold code only as a consequence

```
static void foo() { ... }
static int* bar() { ...; return ...; }
static void baz(int *) { ... }

extern void __attribute__((cold)) sink();
void hotcold(int cond) {
  int *p = ...;
  if (cond) {
    p = bar();
    sink();
    foo();
  }
  baz(p);
}
```

Attributor - Selective Investment

Focus on hot code; look at otherwise cold code only as a consequence

```
static void foo() { ... }
static int* bar() { ...; return ...; }
static void baz(int *) { ... }

extern void __attribute__((cold)) sink();
void hotcold(int cond) {
  int *p = ...;
  if (cond) {
    p = bar();
    sink();
    foo();
  }
  baz(p);
}
```

Attributor - Selective Investment

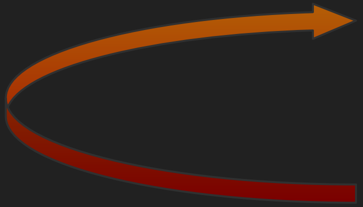
Focus on hot code; look at otherwise cold code only as a consequence

```
static void foo() { ... }
static int* bar() { ...; return ...; }
static void baz(int *) { ... }

extern void __attribute__((cold)) sink();
void hotcold(int cond) {
  int *p = ...;
  if (cond) {
    p = bar();
    sink();
    foo();
  }
  baz(p);
}
```

Attributor - Selective Investment

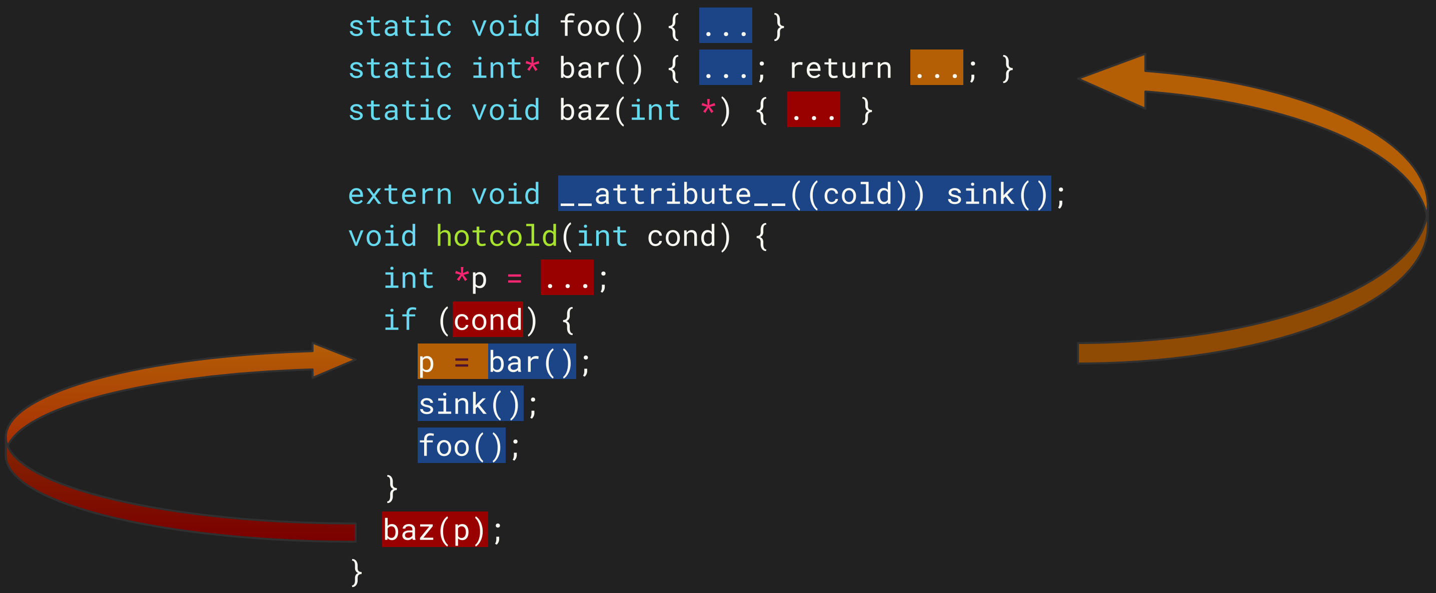
Focus on hot code; look at otherwise cold code only as a consequence

```
static void foo() { ... }
static int* bar() { ...; return ...; }
static void baz(int *) { ... }

extern void __attribute__((cold)) sink();
void hotcold(int cond) {
  int *p = ...;
  if (cond) {
    p = bar();
    sink();
    foo();
  }
  baz(p);
}
```



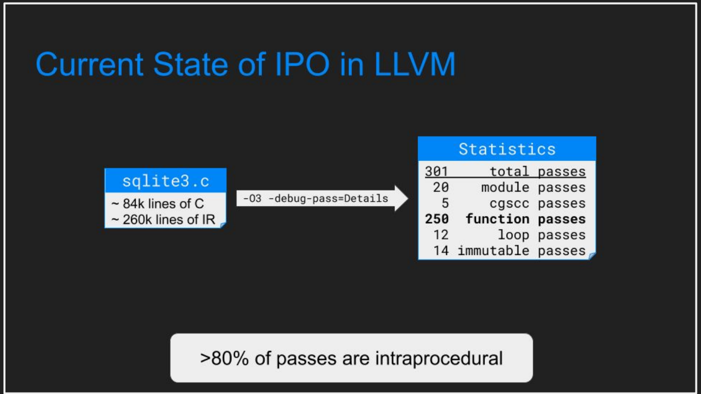
Attributor - Selective Investment

Focus on hot code; look at otherwise cold code only as a consequence



```
static void foo() { ... }
static int* bar() { ...; return ...; }
static void baz(int *) { ... }

extern void __attribute__((cold)) sink();
void hotcold(int cond) {
  int *p = ...;
  if (cond) {
    p = bar();
    sink();
    foo();
  }
  baz(p);
}
```

Conclusions

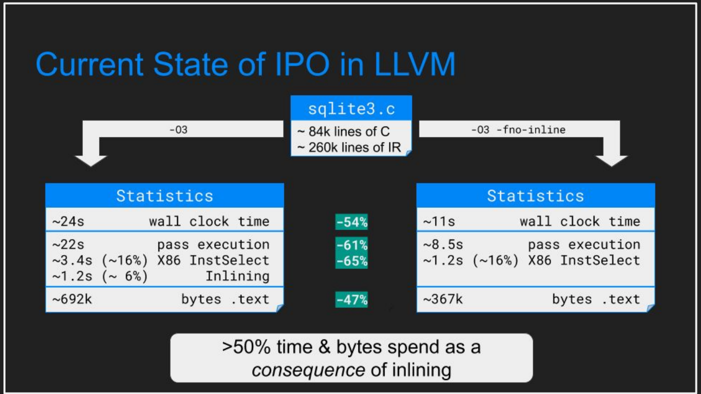


Current State of IPO in LLVM

```
sqlite3.c
~ 84k lines of C
~ 260k lines of IR
```

```
Statistics
301      total passes
 20     module passes
  5      cgscc passes
250   function passes
 12       loop passes
 14  immutable passes
```

>80% of passes are intraprocedural



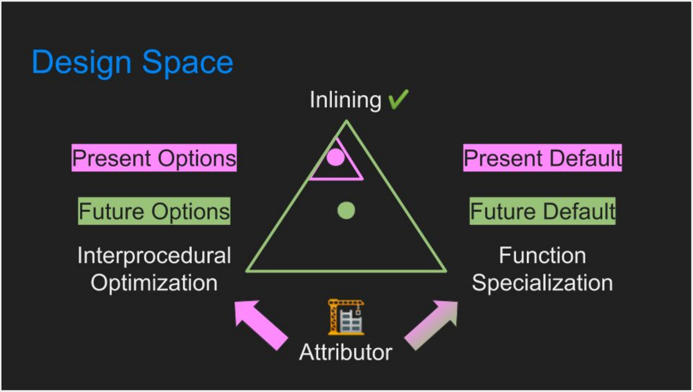
Current State of IPO in LLVM

```
sqlite3.c
~ 84k lines of C
~ 260k lines of IR
```

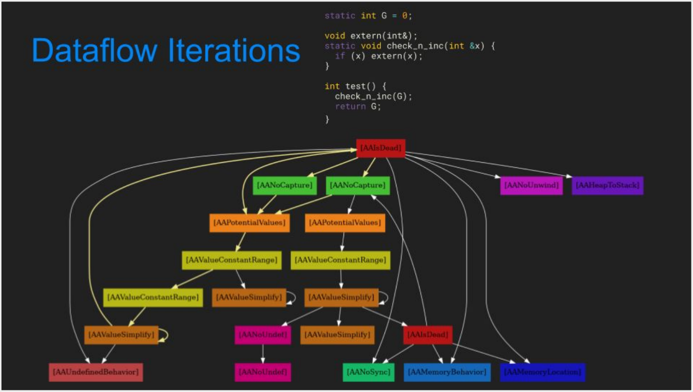
```
Statistics
~24s              wall clock time
~22s               pass execution
~3.4s (~16%)  X86 InstSelect
~1.2s (~ 6%)          Inlining
~692k                bytes .text
```

```
Statistics
~11s              wall clock time
~8.5s              pass execution
~1.2s (~16%)  X86 InstSelect
~367k                bytes .text
```

>50% time & bytes spend as a *consequence* of inlining



Design Space



Dataflow Iterations

```
static int G = 0;

void extern(int&);
static void check_n_inc(int &x) {
  if (x) extern(x);
}

int test() {
  check_n_inc(G);
  return G;
}
```

References

1. Tech talk: The Attributor: A Versatile Inter-procedural Fixpoint, J. Doerfert, S. Stipanovic, H. Ueno, LLVM Developers' Meeting 2019
2. (OpenMP) Parallelism Aware Optimizations, LLVM Developers' Meeting 2020
3. Hot Cold Splitting Optimization Pass In LLVM, A. Kumar, LLVM Developers' Meeting 2019
4. Devirtualization in LLVM, P. Padlewski, LLVM Developers' Meeting 2016
5. A Deep Dive into the Interprocedural Optimization Infrastructure, LLVM Developers' Meeting 2020
6. The Attributor: A Versatile Inter-procedural Fixpoint, J. Doerfert, S. Stipanovic, H. Ueno, LLVM Developers' Meeting 2019
7. ThinLTO: Scalable and Incremental Link-Time Optimization, Teresa Johnson, CppCon 2017
8. Cross-Translation Unit Optimization via Annotated Headers, W. Moses, J. Doerfert, LLVM Developers' Meeting 2019
9. Tutorial: The Attributor: A Versatile Inter-procedural Fixpoint, J. Doerfert, S. Stipanovic, H. Ueno, LLVM Developers' Meeting 2019
10. GCC common function attributes