A whirlwind tour of the LLVM optimizer

Nikita Popov @ EuroLLVM 2023

Red Hat

Agenda

- High-level overview of the middle-end optimization pipeline
- Brief description of important optimization passes
  - Get basic idea about pass responsibilities
  - Learn about key restrictions/constraints

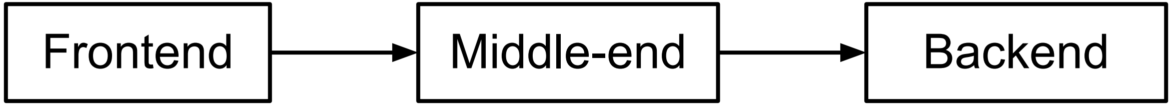
About Me

- Software Engineer on Platform Tools team at Red Hat
  - Packaging of LLVM for Fedora, CentOS and RHEL
  - Upstream work on LLVM and Clang

About Me

- Software Engineer on Platform Tools team at Red Hat
  - Packaging of LLVM for Fedora, CentOS and RHEL
  - Upstream work on LLVM and Clang
- I work on:
  - The LLVM middle-end
  - LLVM / Rust integration
  - Compilation time improvements (LLVM Compile-Time Tracker)

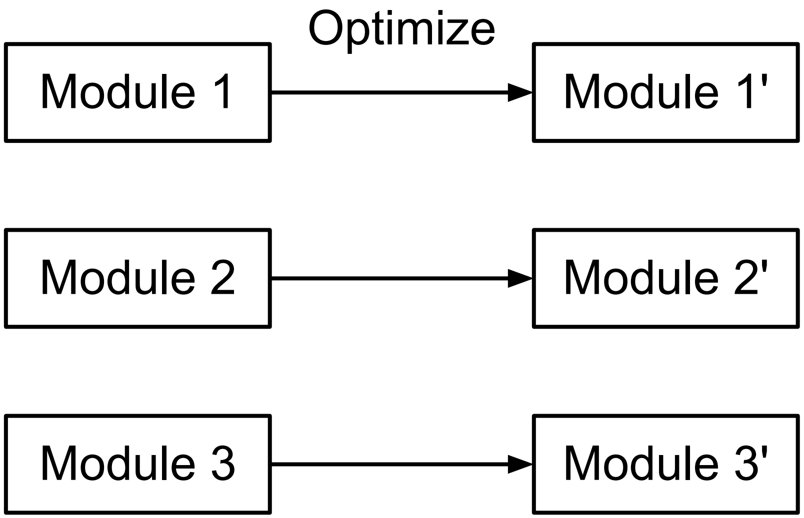
...ends



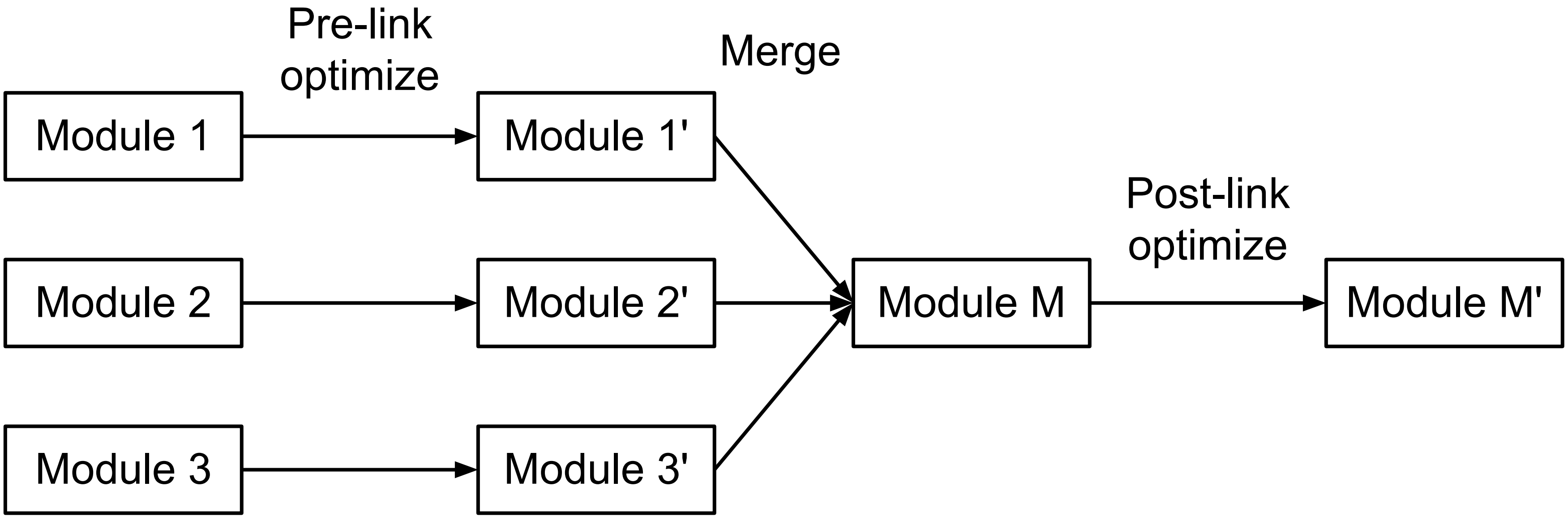
Clang
Rust
Swift
Julia
...

X86
AArch64
ARM
RISCV
...

Default (non-LTO) pipeline

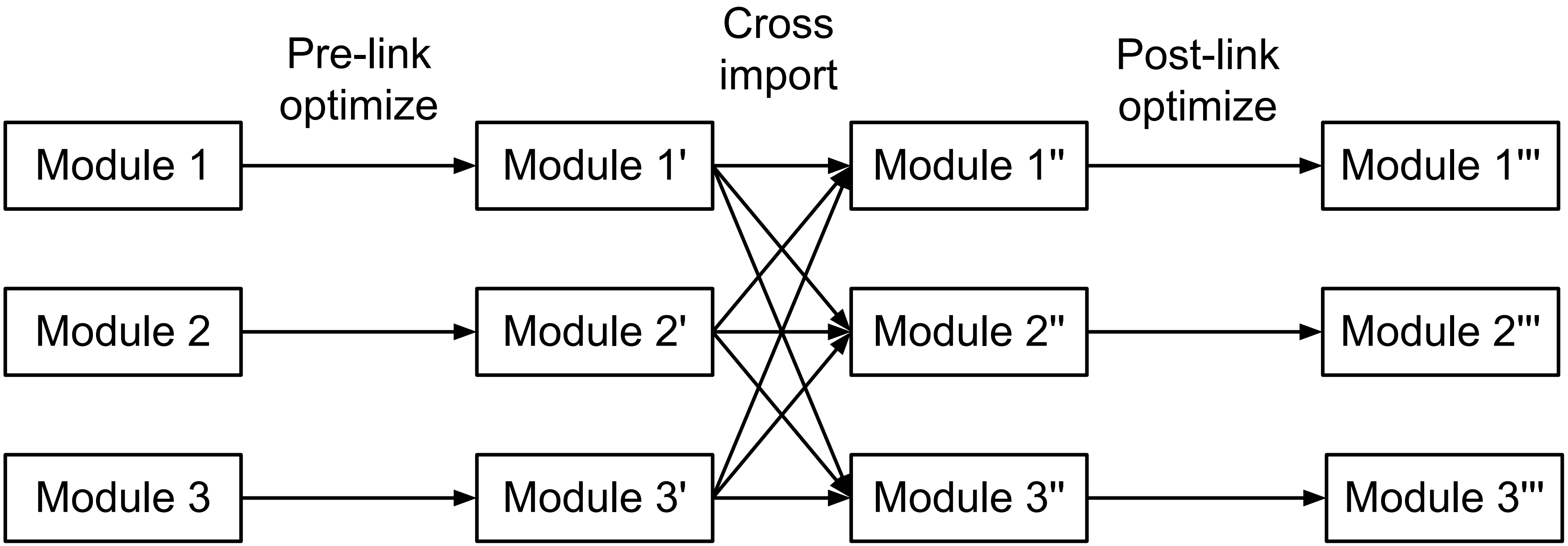


Full LTO pipeline



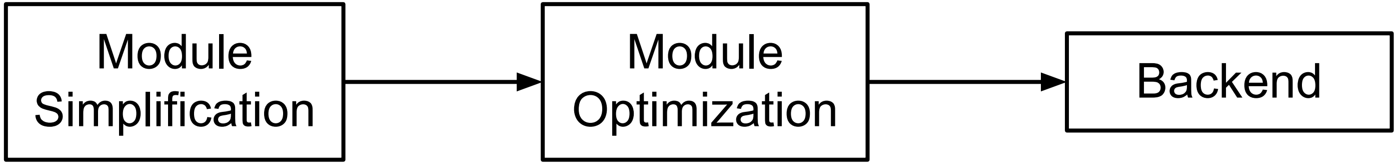
Red Hat

Thin LTO pipeline

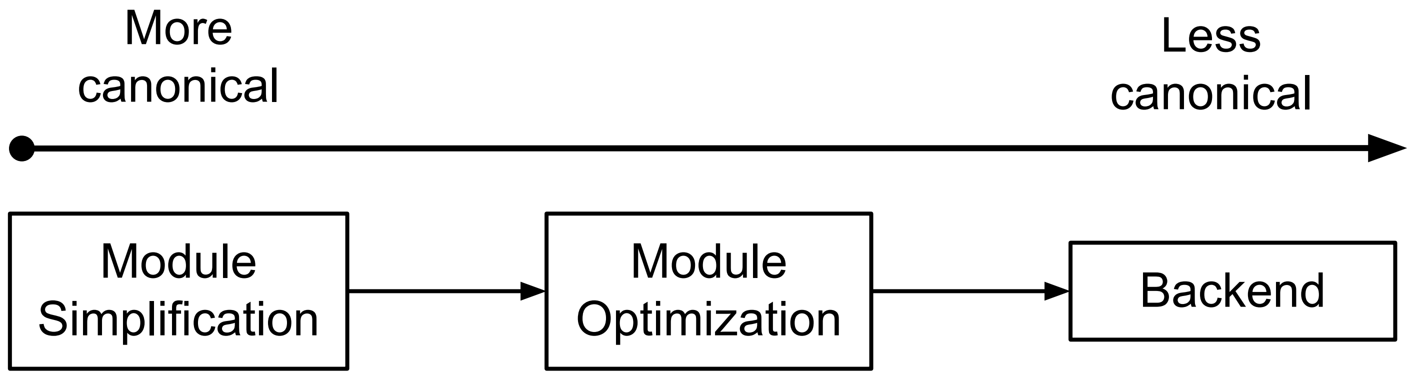


Red Hat

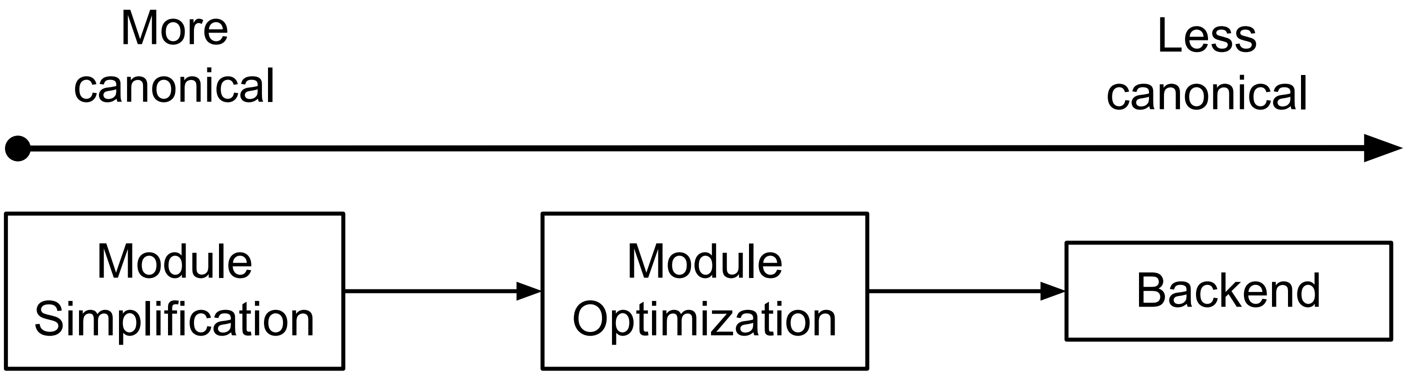
Default pipeline



Default pipeline



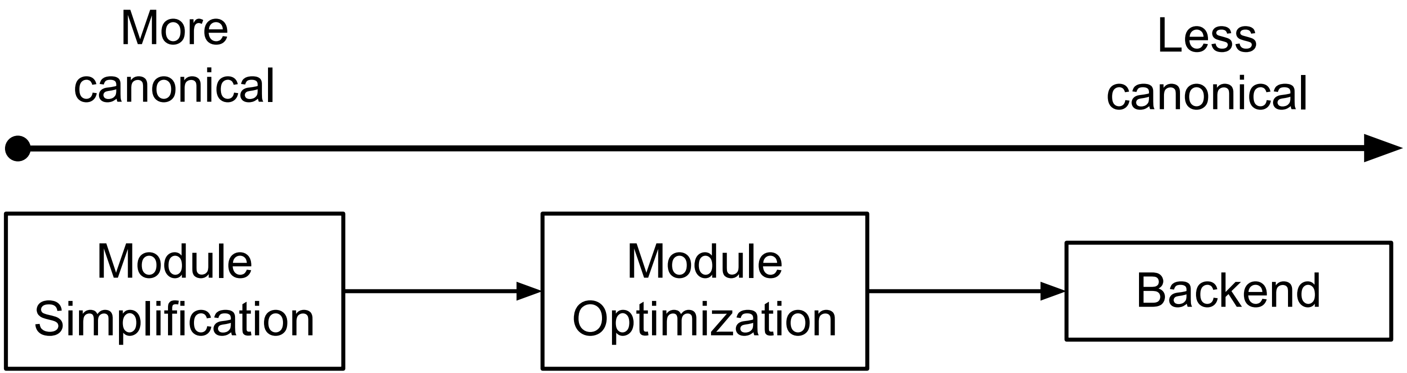
Default pipeline



Inlining
Mem2Reg
LICM (Loop Invariant Code Motion)
...
Make further opts
easier

Red Hat

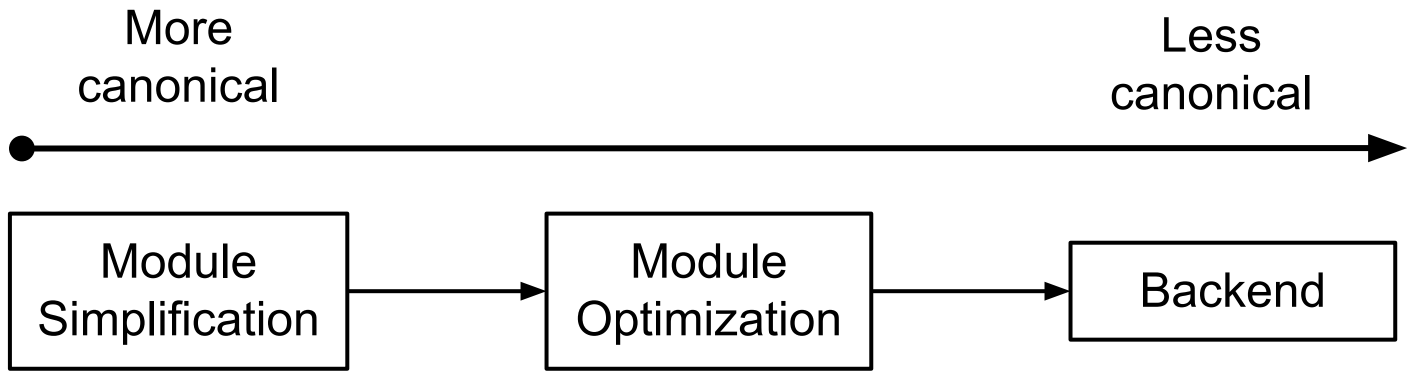
Default pipeline



Inlining
Mem2Reg
LICM
...
Make further opts easier

Vectorization
Runtime unrolling
...
Make further opts harder

Default pipeline



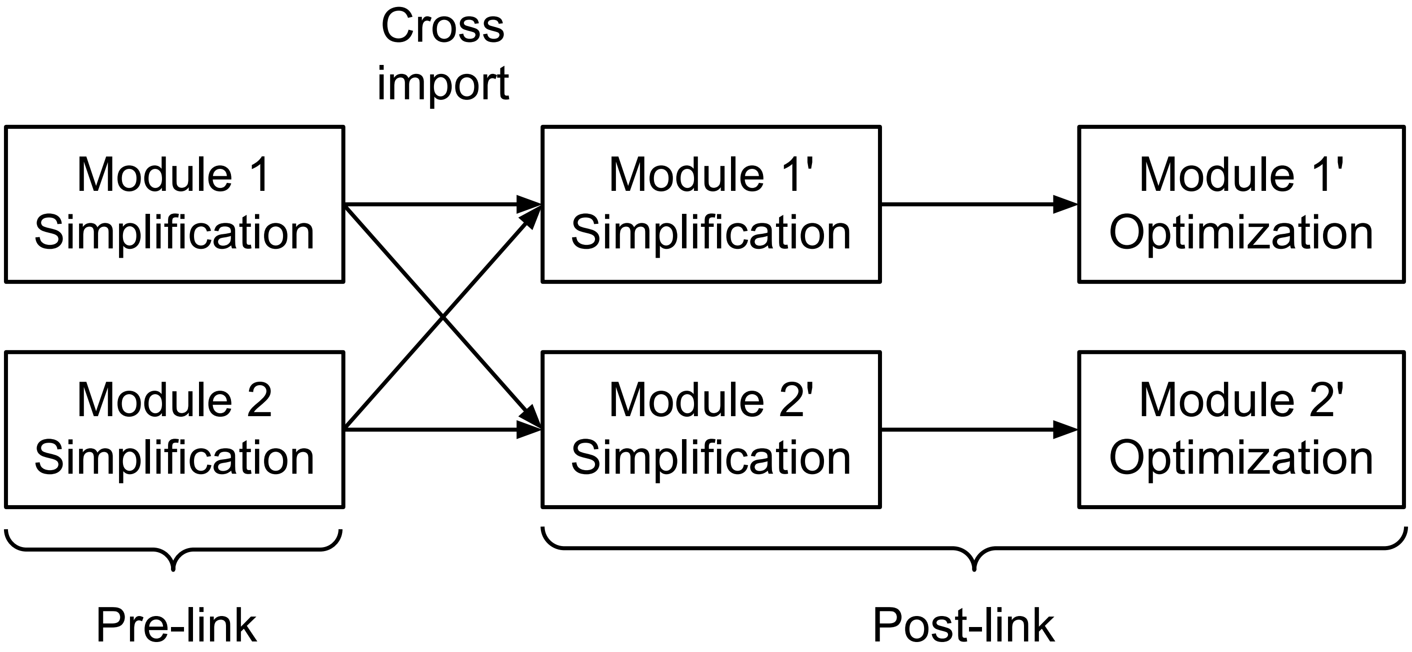
Inlining
Mem2Reg
LICM
...
Make further opts easier

Vectorization
Runtime unrolling
...
Make further opts harder

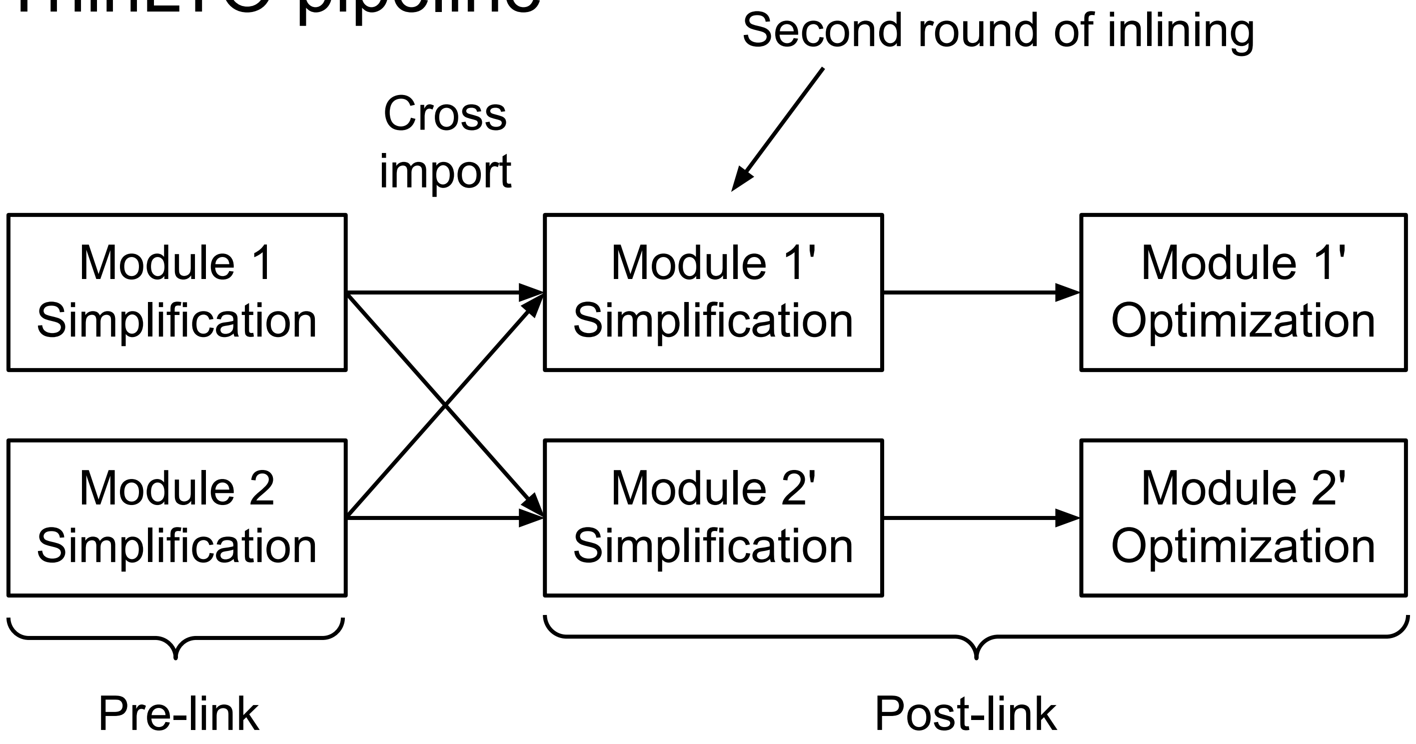
Target-specific optimization

Lowering to machine code

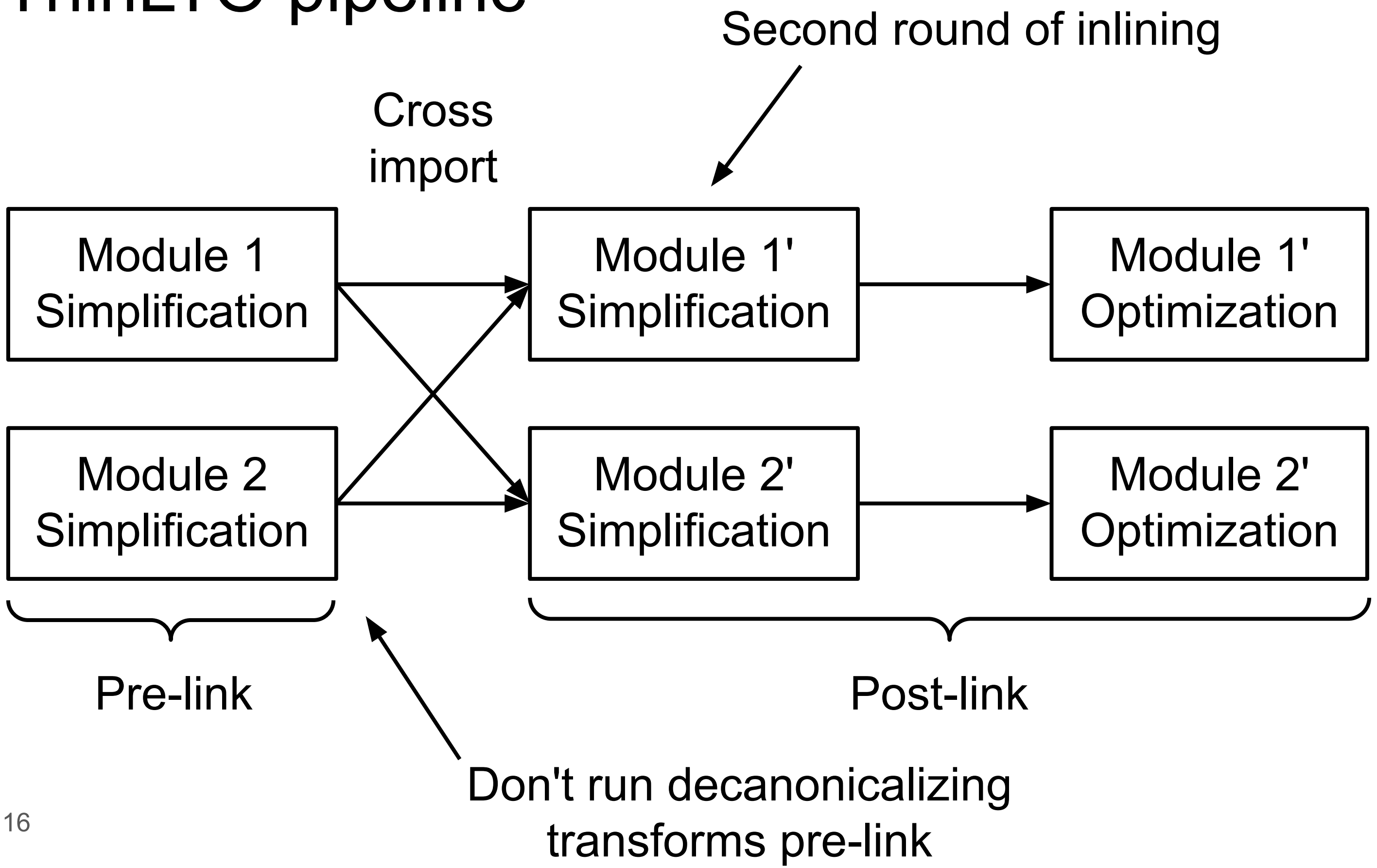
ThinLTO pipeline



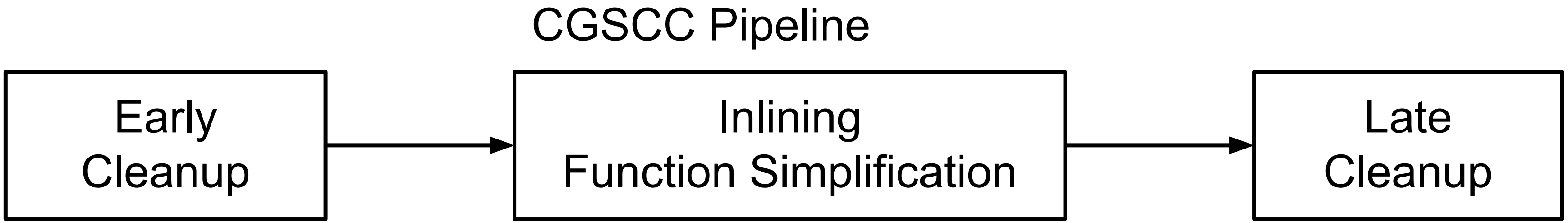
ThinLTO pipeline



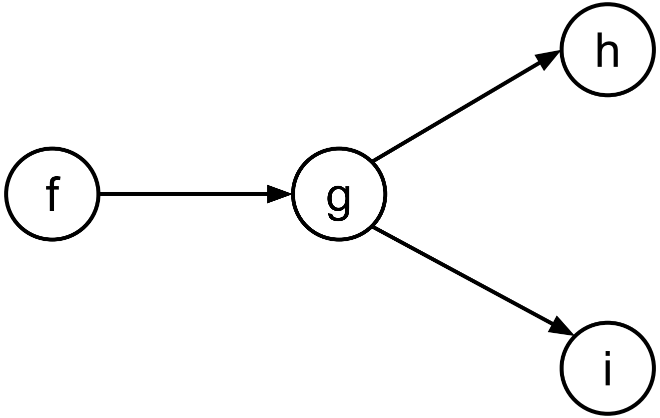
ThinLTO pipeline



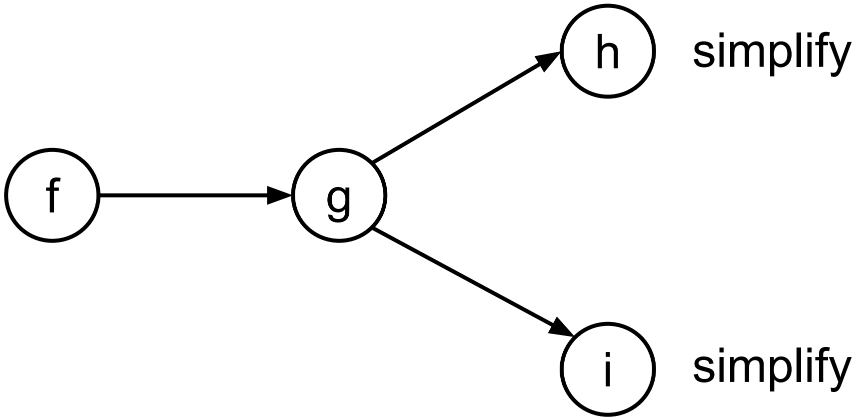
Module Simplification



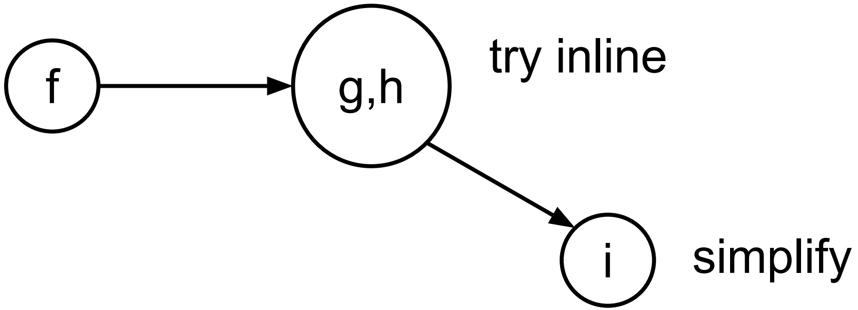
CGSCC Pipeline



CGSCC Pipeline

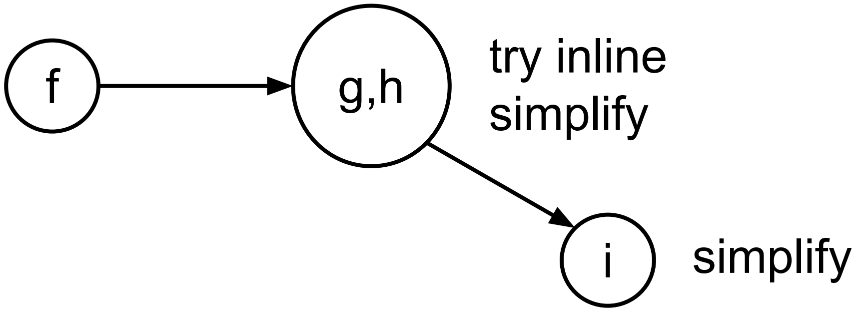


CGSCC Pipeline



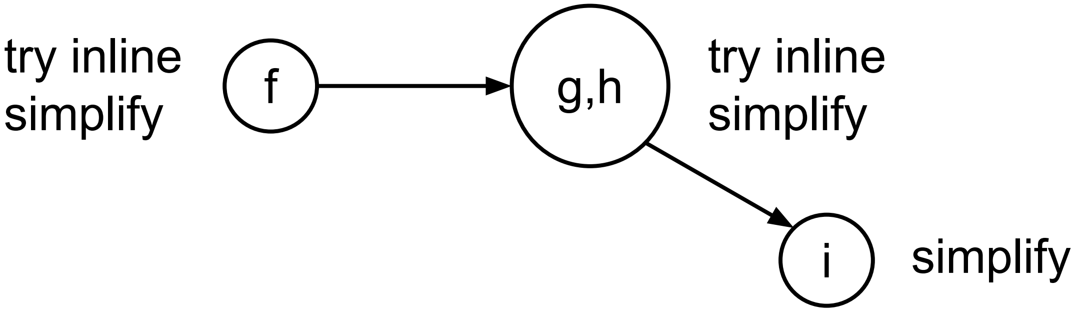
Red Hat

CGSCC Pipeline



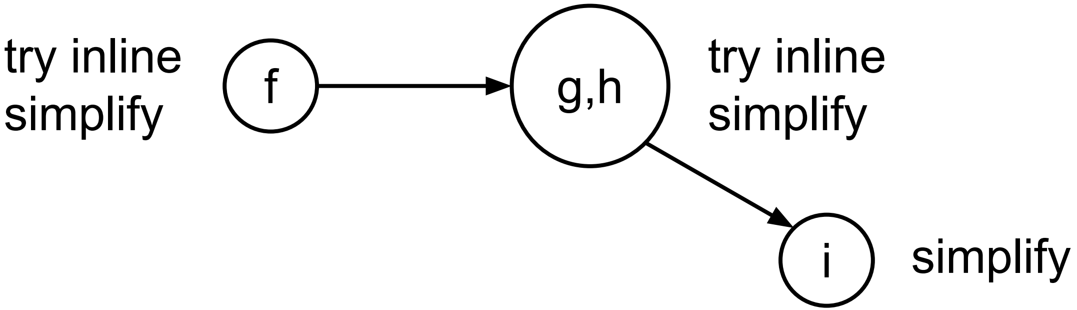
Red Hat

CGSCC Pipeline



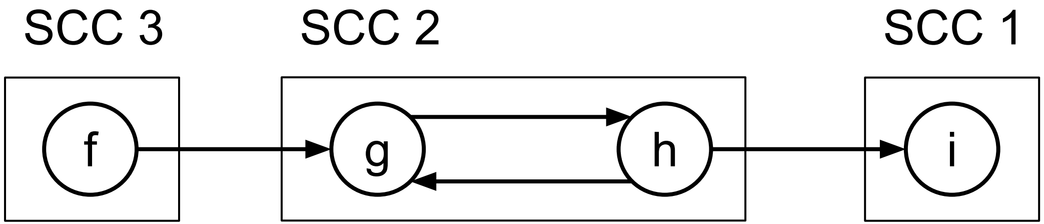
Red Hat

CGSCC Pipeline



Inlining sees already simplified functions!

Call-Graph Strongly Connected Components



No well-defined order within SCC

Running pipelines

- `opt -passes='default<O3>'` == `opt -O3`
- `opt -passes='thinlto-pre-link<O3>'`
- `opt -passes='thinlto<O3>'`
- `opt -passes='lto-pre-link<O3>'`
- `opt -passes='lto<O3>'`

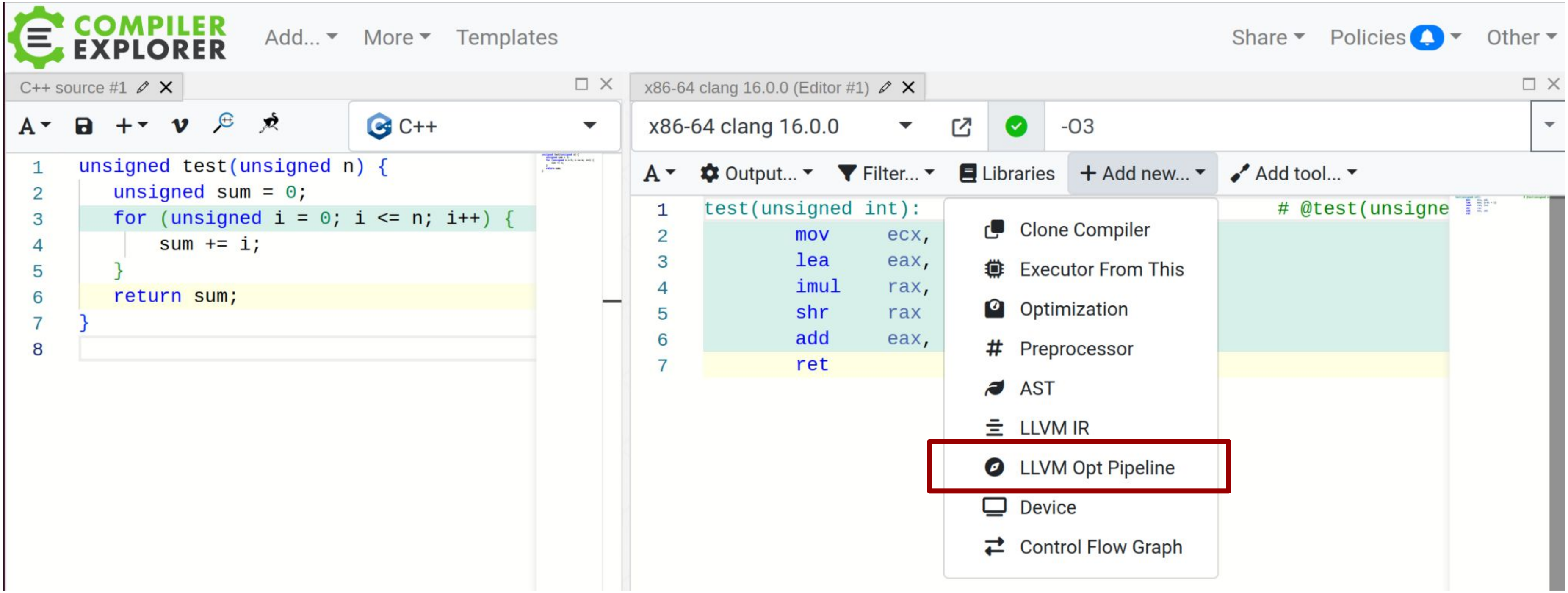
Red Hat

opt -passes='default<O3>' -print-pipeline-passes

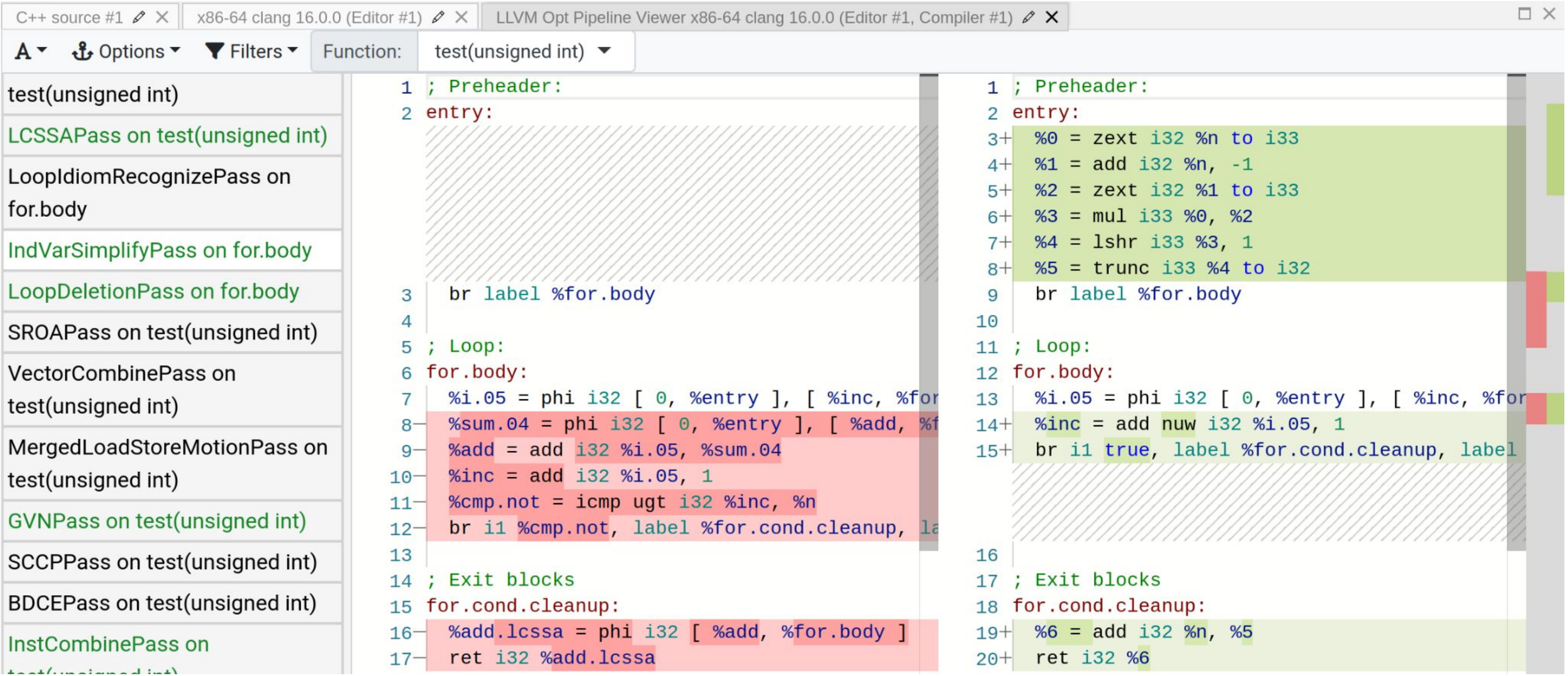
```
annotation2metadata,forceattrs,inferattrs,coro-early,function<eager-inv>(lower-expect,simplifycfg<bonus-inst-threshold=1;no-forw
ard-switch-cond;no-switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts>,sroa<modify-c
fg>,early-cse<>,callsite-splitting),openmp-opt,ipsccp,called-value-propagation,globalopt,function<eager-inv>(mem2reg,instcombine
<max-iterations=1000;no-use-loop-info>,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-
to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts>),require<globals-aa>,function(invalidate<aa>),require<profile-s
ummary>,cgscc(devirt<4>(inline<only-mandatory>,inline,function-attrs<skip-non-recursive>,argpromotion,openmp-opt-cgscc,function<
eager-inv;no-rerun>(sroa<modify-cfg>,early-cse<memssa>,speculative-execution,jump-threading,correlated-propagation,simplifycfg<b
onus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-c
ommon-insts>,instcombine<max-iterations=1000;no-use-loop-info>,aggressive-instcombine,constraint-elimination,libcalls-shrinkwrap
,tailcallelim,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-h
oist-common-insts;no-sink-common-insts>,reassociate,loop-mssa(loop-instsimplify,loop-simplifycfg,licm<no-allowspeculation>,loop-
rotate,licm<allowspeculation>,simple-loop-unswitch<nontrivial;trivial>),simplifycfg<bonus-inst-threshold=1;no-forward-switch-con
d;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts>,instcombine<max-iterations=100
0;no-use-loop-info>,loop(loop-idiom,indvars,loop-deletion,loop-unroll-full),sroa<modify-cfg>,vector-combine,mldst-motion<no-spli
t-footer-bb>,gvn<>,sccp,bdce,instcombine<max-iterations=1000;no-use-loop-info>,jump-threading,correlated-propagation,adce,memcpy
opt,dse,move-auto-init,loop-mssa(licm<allowspeculation>),coro-elide,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;sw
itch-range-to-icmp;no-switch-to-lookup;keep-loops;hoist-common-insts;sink-common-insts>,instcombine<max-iterations=1000;no-use-l
oop-info>),function-attrs,function(require<should-not-run-function-passes>),coro-split)),deadargelim,coro-cleanup,globalopt,glob
aldce,elim-avail-extern,rpo-function-attrs,recompute-globalsaa,function<eager-inv>(float2int,lower-constant-intrinsics,chr,loop(
loop-rotate,loop-deletion),loop-distribute,inject-tli-mappings,loop-vectorize<no-interleave-forced-only;no-vectorize-forced-only
;>,loop-load-elim,instcombine<max-iterations=1000;no-use-loop-info>,simplifycfg<bonus-inst-threshold=1;forward-switch-cond;switc
h-range-to-icmp;switch-to-lookup;no-keep-loops;hoist-common-insts;sink-common-insts>,slp-vectorizer,vector-combine,instcombine<m
ax-iterations=1000;no-use-loop-info>,loop-unroll<O3>,transform-warning,sroa<preserve-cfg>,instcombine<max-iterations=1000;no-use
-loop-info>,loop-mssa(licm<allowspeculation>),alignment-from-assumptions,loop-sink,instsimplify,div-rem-pairs,tailcallelim,simpl
ifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;n
o-sink-common-insts>),globaldce,constmerge,cg-profile,rel-lookup-table-converter,function(annotation-remarks),verify,print
```

Defined in `PassBuilderPipelines.cpp`

godbolt.org – LLVM Opt Pipeline



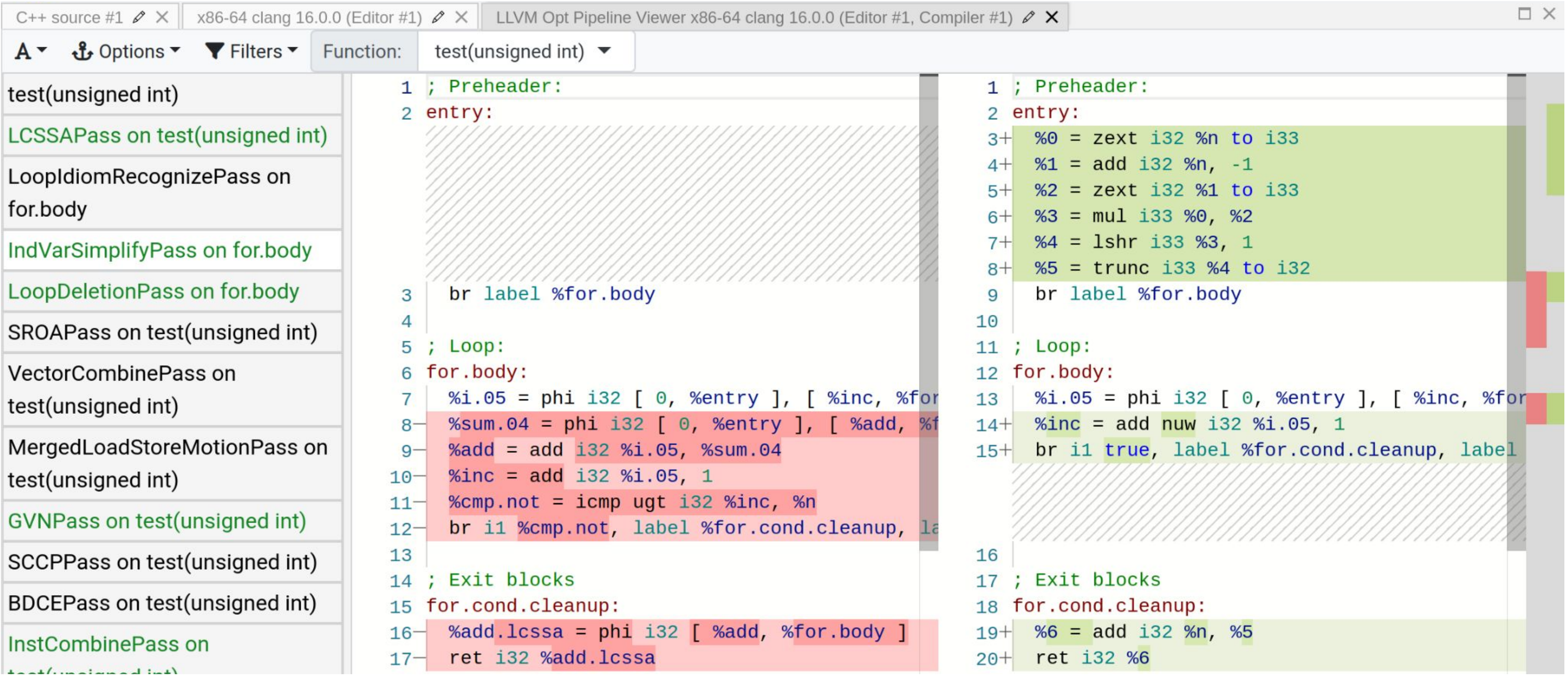
godbolt.org – LLVM Opt Pipeline



Red Hat

godbolt.org – LLVM Opt Pipeline

Or run `opt -print-after-all` locally



Red Hat

SSA Construction

Mem2Reg

```
int test(int x, int y) {
    return x + y;
}
```

Mem2Reg

```
define i32 @test(i32 %x, i32 %y) {
entry:
 %x.addr = alloca i32
 %y.addr = alloca i32
 store i32 %x, ptr %x.addr
 store i32 %y, ptr %y.addr
 %0 = load i32, ptr %x.addr
 %1 = load i32, ptr %y.addr
 %add = add nsw i32 %0, %1
 ret i32 %add
}
```

Mem2Reg

```
define i32 @test(i32 %x, i32 %y) {
entry:
 %add = add nsw i32 %x, %y
 ret i32 %add
}
```

SROA: Scalar Replacement of Aggregates

- Break up allocas into smaller allocas based on access pattern
  - `%vec = alloca { ptr, i64, i64 }`
  - `-> %vec.ptr = alloca ptr`
  - `-> %vec.size = alloca i64`
  - `-> %vec.capacity = alloca i64`

SROA: Scalar Replacement of Aggregates

- Break up allocas into smaller allocas based on access pattern
  - `%vec = alloca { ptr, i64, i64 }`
  - `-> %vec.ptr = alloca ptr`
  - `-> %vec.size = alloca i64`
  - `-> %vec.capacity = alloca i64`
- Then run Mem2Reg to convert alloca/load/store to SSA values

SROA: Scalar Replacement of Aggregates

- Break up allocas into smaller allocas based on access pattern
  - `%vec = alloca { ptr, i64, i64 }`
  - `-> %vec.ptr = alloca ptr`
  - `-> %vec.size = alloca i64`
  - `-> %vec.capacity = alloca i64`
- Then run Mem2Reg to convert alloca/load/store to SSA values
- Knows many tricks for overlapping accesses
  - For example inserting/extracting bits of a larger integer

Control-Flow Optimization

SimplifyCFG

- The kitchen sink of control-flow transforms
  - If it fits nowhere else, put it here!

SimplifyCFG: Hoist

```
if (cond) {
  foo();
  a();
} else {
  foo();
  b();
}
```

→

```
foo();
if (cond) {

  a();
} else {

  b();
}
```


Red Hat

SimplifyCFG: Speculate

```
if (cond) {
  x = foo();
} else {
  x = 0;
}
```

→

```
tmp = foo();
x = cond ? tmp : 0;
```

SimplifyCFG: Switch to lookup table

```
switch (x) {
case 0:
  return 10;
case 1:
  return 42;
case 2:
  return 123;
case 3:
  return 7;
default:
  return 13;
}
```

→

```
int table[] = {10, 42, 123, 7};

if (x < 4) {
  return table[x];
} else {
  return 13;
}
```

SimplifyCFG

- The kitchen sink of control-flow transforms
  - If it fits nowhere else, put it here!
- Invoked with many different options at different pipeline positions
  - Some transforms only run late in the pipeline

SimplifyCFG

- The kitchen sink of control-flow transforms
  - If it fits nowhere else, put it here!
- Invoked with many different options at different pipeline positions
  - Some transforms only run late in the pipeline
- Can use target-dependent cost model (via TargetTransformInfo)

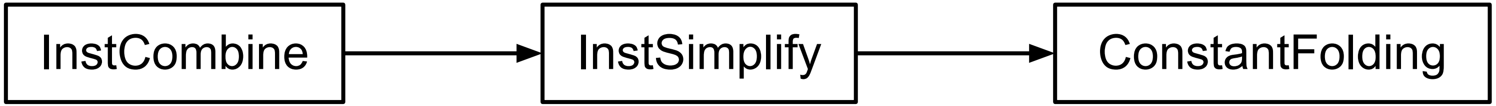
Instruction Combining

(Peephole Optimization)

InstCombine

- The kitchen sink of non-CFG transforms
  - If it fits nowhere else, put it here!

InstCombine: Analysis helpers



InstCombine: Analysis helpers

- ConstantFolding
  - Folds instructions with constant operands to constants
  - 1 + 2 => 3

InstCombine: Analysis helpers

- ConstantFolding
  - Folds instructions with constant operands to constants
  - `1 + 2   =>   3`
- InstSimplify
  - Folds instructions to existing values or constants
  - `x + 0   =>   x`
  - `x - x   =>   0`

InstCombine: Analysis helpers

- ConstantFolding
  - Folds instructions with constant operands to constants
  - `1 + 2   =>   3`
- InstSimplify
  - Folds instructions to existing values or constants
  - `x + 0   =>   x`
  - `x - x   =>   0`
- InstCombine
  - Tries constant folding and instruction simplification first
  - Performs folds that create or modify instructions
  - `x * 4   =>   x << 2`

InstCombine

- The kitchen sink of non-CFG transforms
  - If it fits nowhere else, put it here!
  - Use InstSimplify / ConstantFolding for transforms that don't create/modify instructions.

InstCombine

- The kitchen sink of non-CFG transforms
    - If it fits nowhere else, put it here!
    - Use InstSimplify / ConstantFolding for transforms that don't create/modify instructions.
- Also used to paper over phase ordering issues
    - InstCombine re-implements weak versions of transforms from other passes
    - For example: Basic store-to-load forwarding (usually done by EarlyCSE/GVN)

…Combine

- InstCombine
  - Canonicalization pass: **Cannot** be target-dependent
  - Backend implements reverse/undo transform if necessary

…Combine

- InstCombine
  - Canonicalization pass: **Cannot** be target-dependent
  - Backend implements reverse/undo transform if necessary
- AggressiveInstCombine
  - For expensive transforms, only runs once in pipeline
  - Target-dependence discouraged but sometimes allowed

…Combine

- InstCombine
  - Canonicalization pass: **Cannot** be target-dependent
  - Backend implements reverse/undo transform if necessary
- AggressiveInstCombine
  - For expensive transforms, only runs once in pipeline
  - Target-dependence discouraged but sometimes allowed
- VectorCombine
  - For target-dependent, cost-model driven vector transforms

CVP: CorrelatedValuePropagation

- Optimizations based on value range information (from LazyValueInfo)
- Important for bounds check elimination
  - `icmp ult i32 %x, 10  =>  i1 true if %x in [0, 10)`

CVP: CorrelatedValuePropagation

- Optimizations based on value range information (from LazyValueInfo)
- Important for bounds check elimination
  - `icmp ult i32 %x, 10  =>  i1 true` if `%x` in `[0, 10)`
- Other range based optimizations
  - `sdiv i32 %x, %y  =>  udiv i32 %x, %y` if `%x`, `%y` non-negative

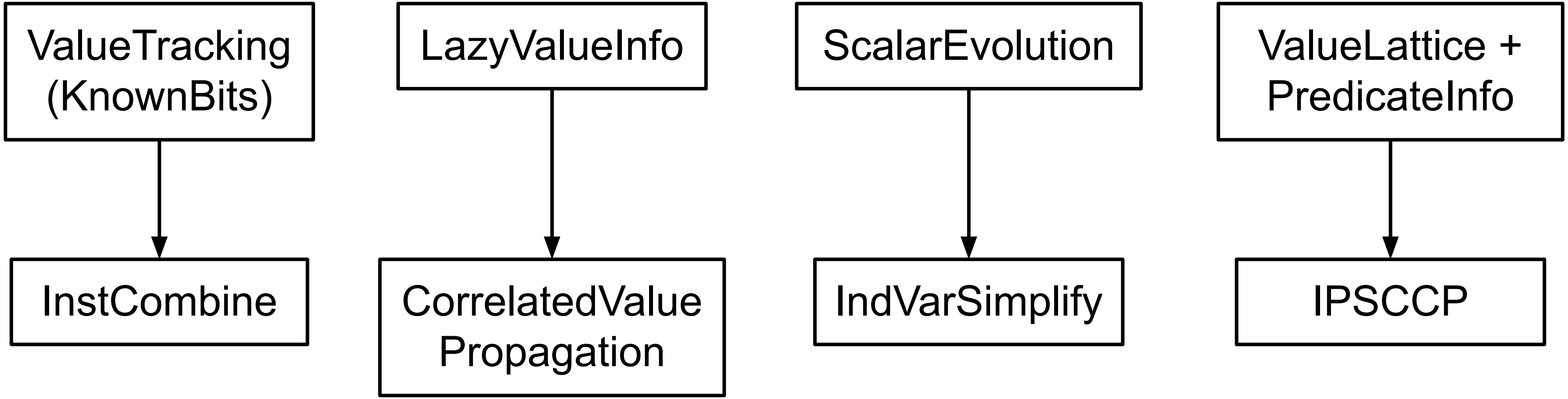
Red Hat

Same transform, different analysis

- Some folds (e.g. sdiv -> udiv) are implemented in multiple passes
    - Folds are driven by different analyses, which are good at different things

Same transform, different analysis

- Some folds (e.g. sdiv -> udiv) are implemented in multiple passes
  - Folds are driven by different analyses, which are good at different things



Redundancy Elimination

EarlyCSE: Common Subexpression Elimination

```
add1 = x + y;
// ...
add2 = x + y;

use(add1);
use(add2);
```

→

```
add1 = x + y;
// ...


use(add1);
use(add1);
```


Red Hat

EarlyCSE: Common Subexpression Elimination

- Basic CSE based on scoped hash table
- Load CSE and store-to-load forwarding using MemorySSA

EarlyCSE: Store to load forwarding

```
*p = v1;
// p not written here
v2 = *p;

use(v1);
use(v2);
```

→

```
*p = v1;



use(v1);
use(v1);
```


Red Hat

GVN: Global Value Numbering

- More general (and much more expensive!) than EarlyCSE
- Uses MemoryDependenceAnalysis
- Non-local load CSE
- Partial redundancy elimination (PRE)

GVN: Non-local load CSE

```
if (...) {
   v1 = *p;
} else {
   *p = v2;
}

v3 = *p;
use(v3);
```

→

```
if (...) {
   v1 = *p;
} else {
   *p = v2;
}

v3 = phi(v1, v2);
use(v3);
```

GVN: Load PRE

```
if (...) {

} else {
    *p = v1;
}

v2 = *p;
use(v2);
```

→

```
if (...) {
    v2_pre = *p;
} else {
    *p = v1;
}

v2 = phi(v2_pre, v1);
use(v2);
```


Red Hat

Memory Optimizations

MemCpyOpt

- Optimize memcpy and memset using MemorySSA

MemCpyOpt: Memcpy forwarding

```
memcpy(y, x, 16);
// y not written here
memcpy(z, y, 16);
```

→

```
memcpy(y, x, 16);
// y not written here
memcpy(z, x, 16);
```

MemCpyOpt: Call Slot Optimization

```
Ty tmp;
foo(tmp);
memcpy(dst, tmp, sizeof(Ty));
```

→

```
foo(dst);
```

DSE: Dead Store Elimination

- Remove dead stores using MemorySSA

DSE: Dead Store Elimination

```
*p = v1;
// p not read here
*p = v2;
```

→

```
// p not read here
*p = v2;
```

DSE: Dead before return

```
%p = alloca i32
; ...
store i32 %v, ptr %p
; %p not read here
ret void
```

→

```
%p = alloca i32
; ...

; %p not read here
ret void
```

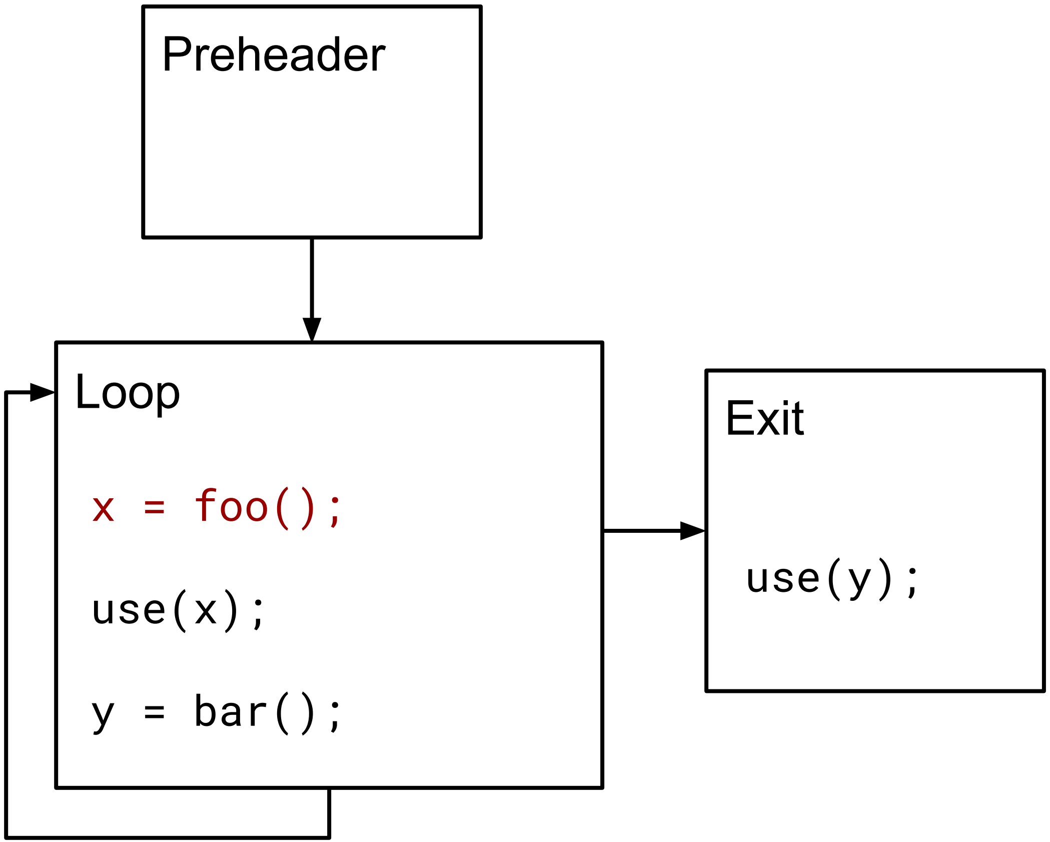

Red Hat

Loop Optimization

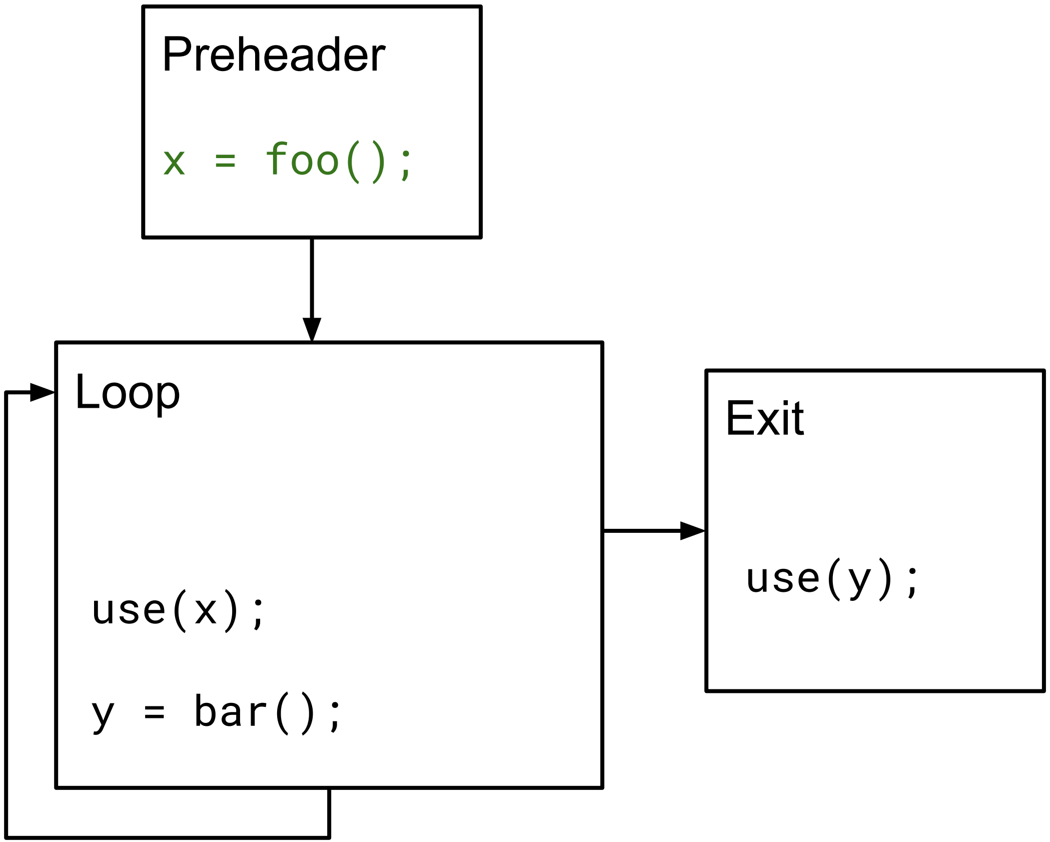
Loop pass manager

- Visit child loops first, then parent loops
- Constructs LoopSimplify and LCSSA (Loop-Closed SSA) form before running

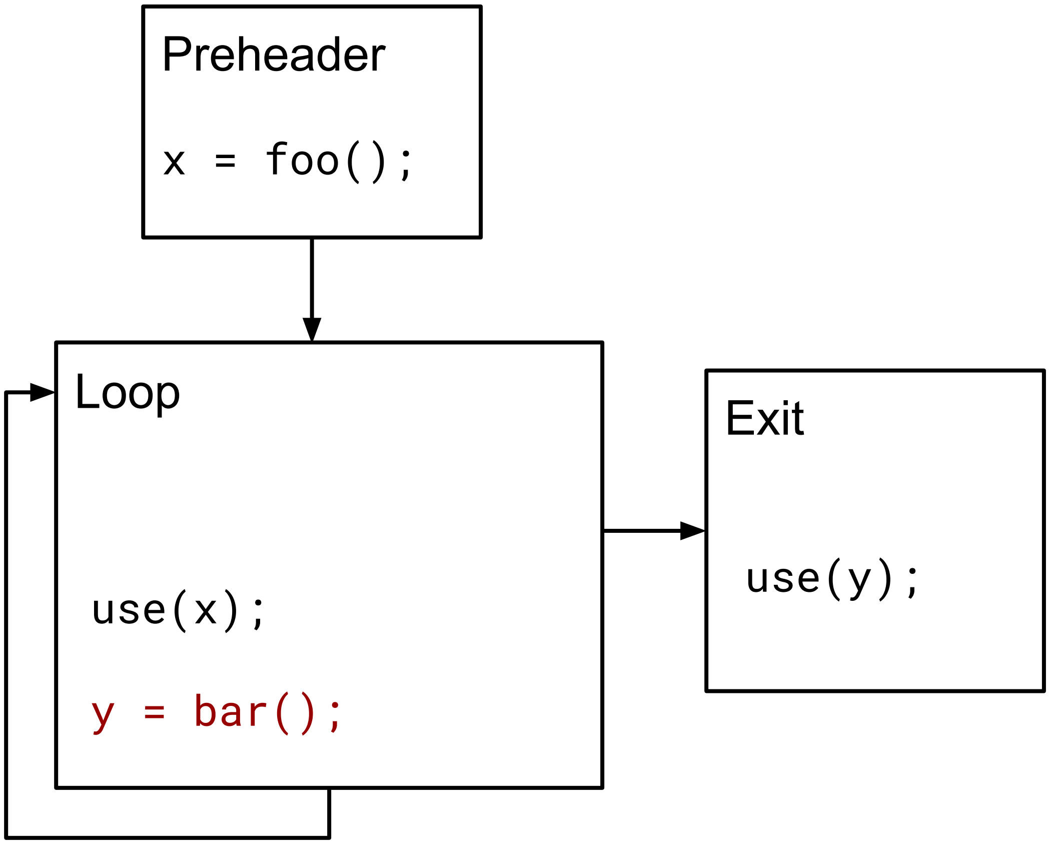
LICM: Hoist



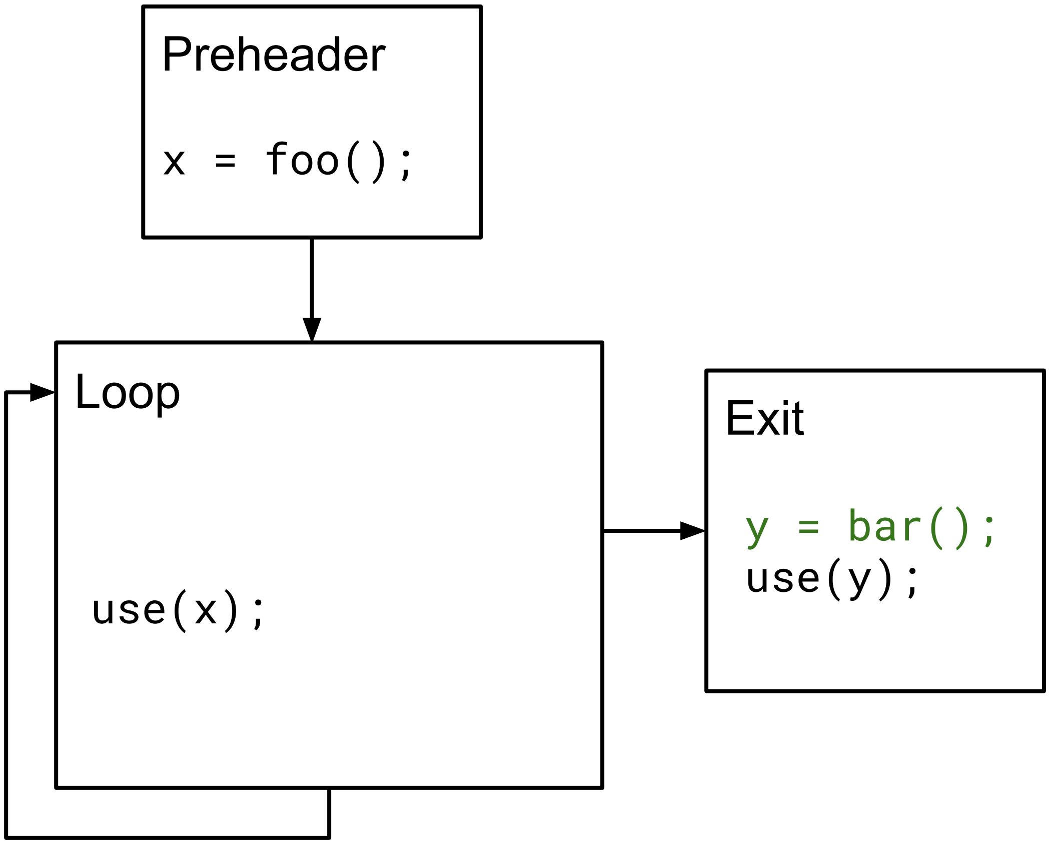
LICM: Hoist



LICM: Sink

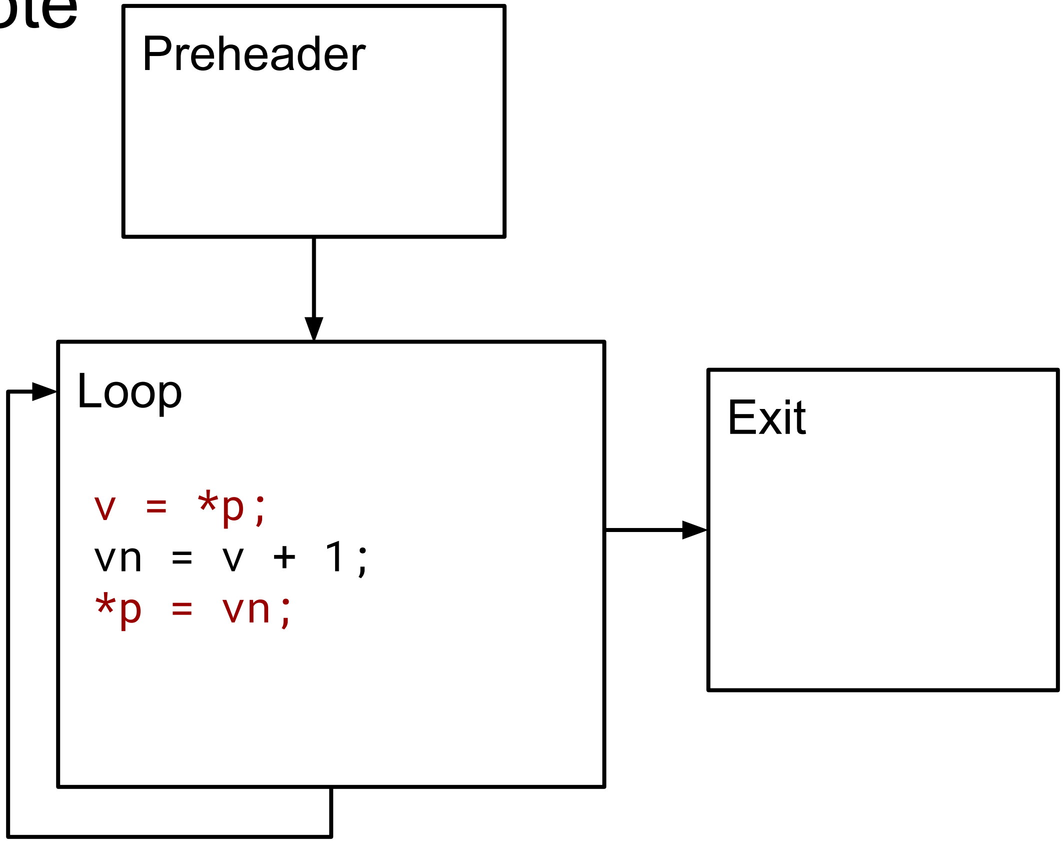


LICM: Sink

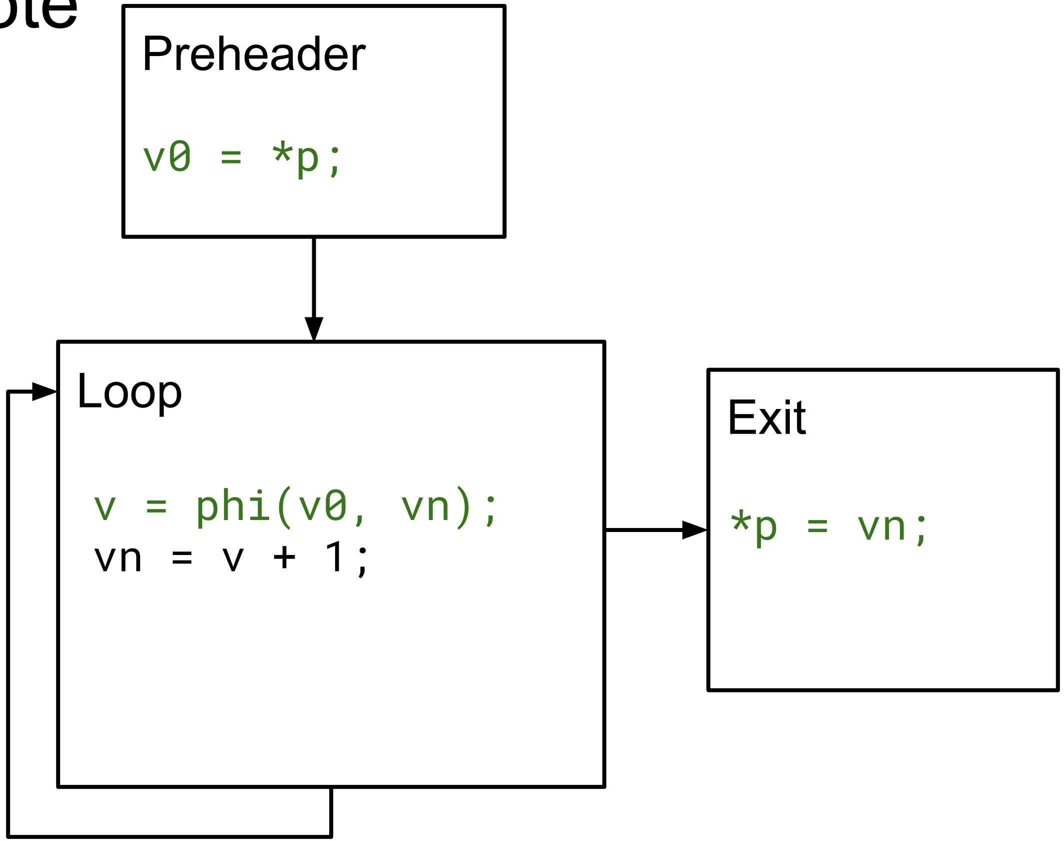


Red Hat

LICM: Promote



LICM: Promote



LICM: Loop Invariant Code Motion

- Transforms:
  - Hoist instructions into preheader
  - Sink instructions into exits
  - Promote scalars
- Uses MemorySSA
- Canonicalization pass: **Cannot** be target or PGO dependent
  - May be undone by LoopSink or MachineSink

IndVarSimplify

- Uses ScalarEvolution analysis
- Simplify induction variables (IVs) and their uses
- Simplify loop exit conditions

IndVarSimplify: Loop exit value replacement

```
unsigned test(unsigned n) {
   unsigned sum = 0;
   for (unsigned i = 0; i <= n; i++) {
       sum += i;
   }
   return sum;
}
```

IndVarSimplify: Loop exit value replacement

```
unsigned test(unsigned n) {
   unsigned sum = 0;
   for (unsigned i = 0; i <= n; i++) {
       sum += i;
   }
   return sum;
}
```

↓

```
unsigned test(unsigned n) {
   for (unsigned i = 0; i <= n; i++) {}

   return (n * (n - 1))/2 + n;
}
```

IndVarSimplify: Loop exit value replacement

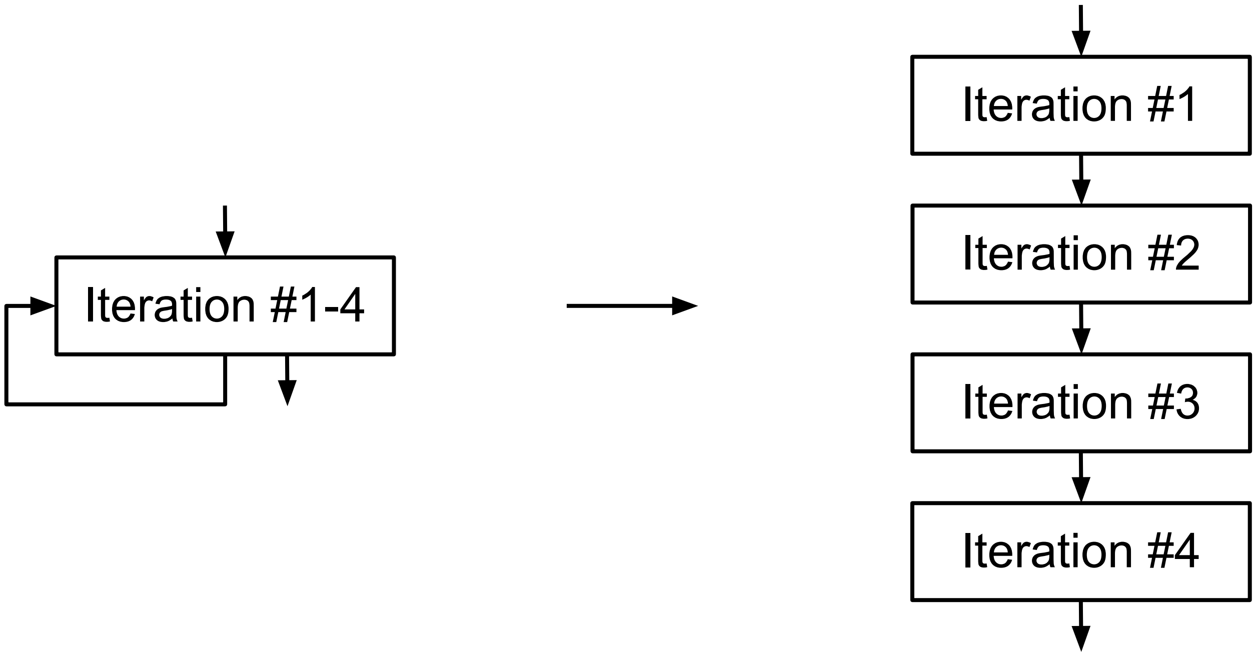
```
unsigned test(unsigned n) {
   unsigned sum = 0;
   for (unsigned i = 0; i <= n; i++) {
       sum += i;
   }
   return sum;
}
```

↓

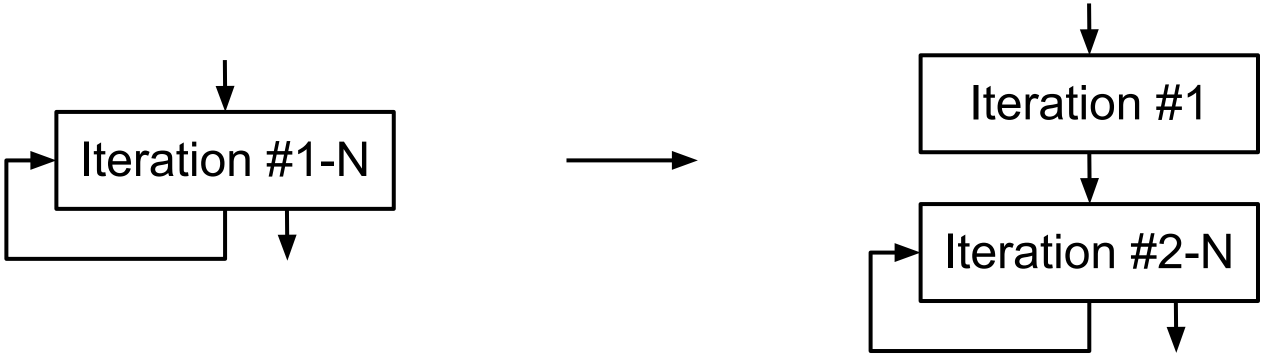
```
unsigned test(unsigned n) {
   for (unsigned i = 0; i <= n; i++) {}

   return (n * (n - 1))/2 + n;
}
```

Later removed by LoopDeletion

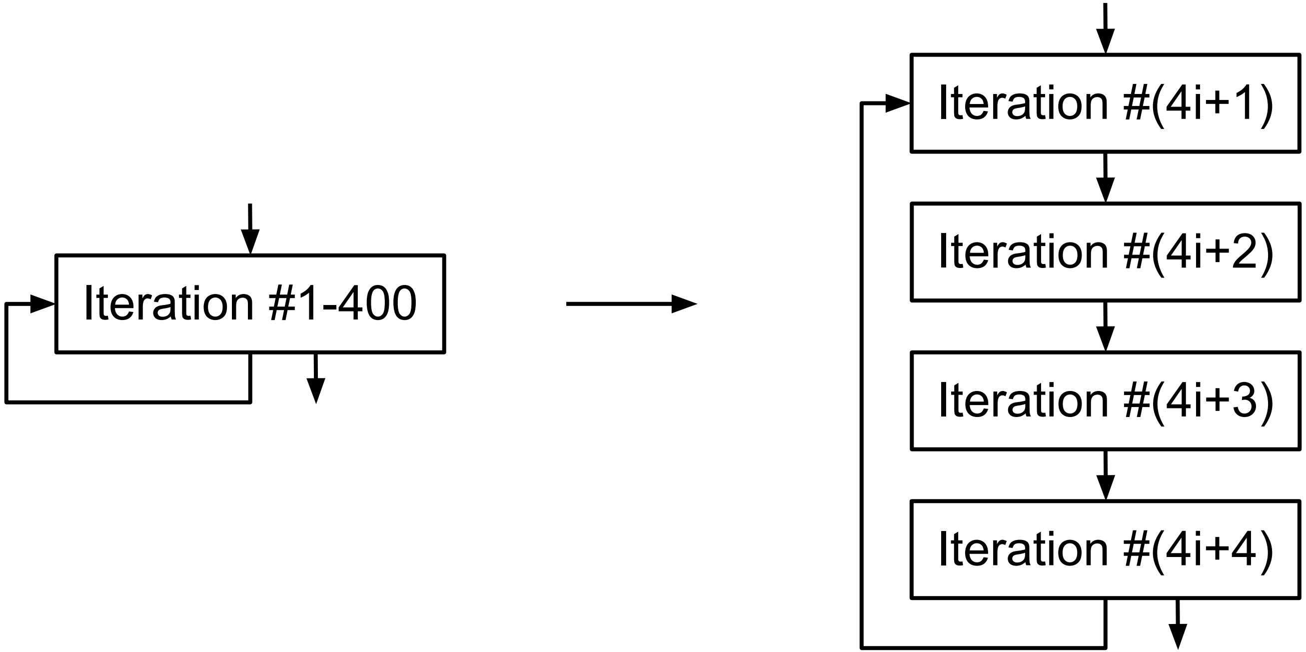
LoopUnroll: Full unrolling



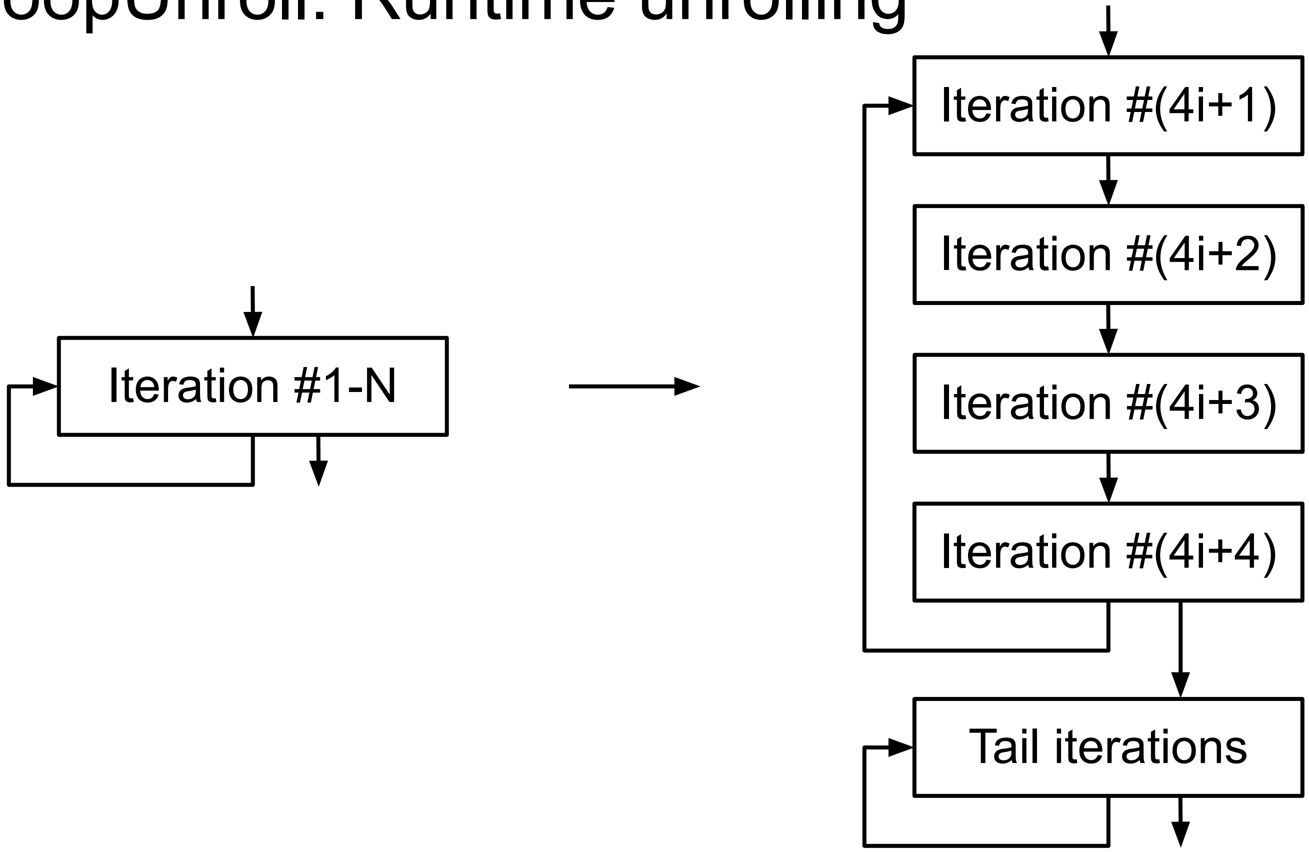
LoopUnroll: Loop peeling



LoopUnroll: Partial unrolling



LoopUnroll: Runtime unrolling



LoopUnroll

- Simplification:
  - Full unrolling (requires known constant trip count)
  - Loop peeling
- Optimization:
  - Partial unrolling (requires known constant trip count/multiple)
  - Runtime unrolling

Vectorization

LoopVectorize

- VPlan to model vectorization without IR changes
- LoopAccessAnalysis to ensure memory dependences are safe
- May require inserting runtime checks and LoopVersioning

SLPVectorize

- SLP = Superword-Level Parallelism
- Vectorizes straight-line code

Inter-Procedural Optimization (IPO)

FunctionAttrs

- Infer attributes on function, arguments and return values
    - nounwind, readonly, nonnull, etc.

FunctionAttrs

- Infer attributes on function, arguments and return values
  - nounwind, readonly, nonnull, etc.
- General approach:
  - Optimistically all functions in the SCC are nounwind
  - Check whether there are any non-nounwind instructions
  - If not, mark all functions in the SCC nounwind

FunctionAttrs

- Infer attributes on function, arguments and return values
  - nounwind, readonly, nonnull, etc.
- General approach:
  - Optimistically all functions in the SCC are nounwind
  - Check whether there are any non-nounwind instructions
  - If not, mark all functions in the SCC nounwind
- New "Attributor" implements much stronger version of this, but not enabled by default (too slow)

IPSCCP: Inter-Procedural Sparse Conditional Constant Propagation

- Propagates constants and constant ranges across functions
- Uses PredicateInfo to take branch conditions into account

IPSCCP: Inter-Procedural Sparse Conditional Constant Propagation

- Propagates constants and constant ranges across functions
- Uses PredicateInfo to take branch conditions into account
- Runs very early, before most simplification (which may lose information)

IPSCCP: Inter-Procedural Sparse Conditional Constant Propagation

- Propagates constants and constant ranges across functions
- Uses PredicateInfo to take branch conditions into account
- Runs very early, before most simplification (which may lose information)
- Also does function specialization (since recently)

Thank You!
Questions?

The End

- Blog: https://www.npopov.com/
- Reach me at:
  - npopov@redhat.com
  - https://twitter.com/nikita_ppv

Bonus Slides

JumpThreading

```
if (x > 10) {
  greater10();
}
always();
if (x > 0) {
  greater0();
}
```

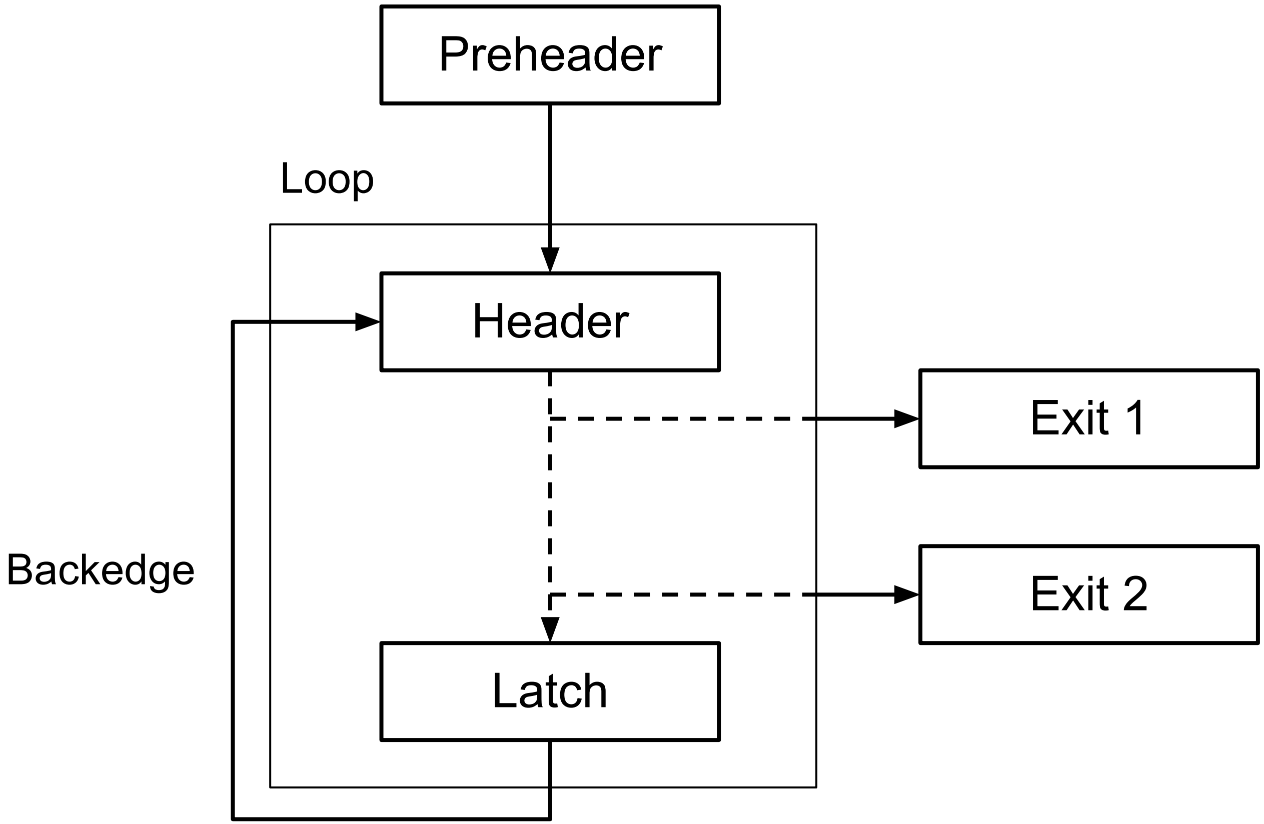
→

```
if (x > 10) {
  greater10();
  always();
  greater0();
} else {
  always();
}
```

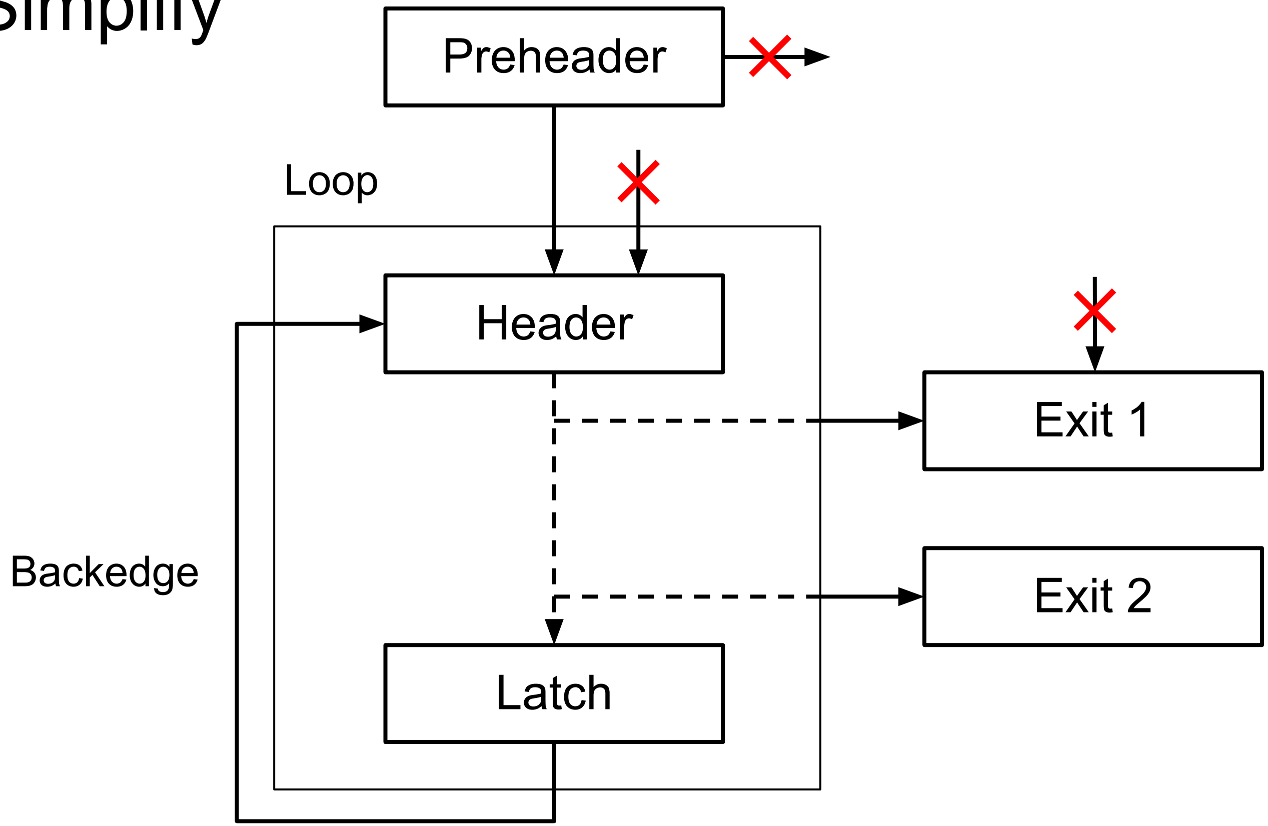
JumpThreading

- Optimizes conditional branches where one condition implies another
- Uses LazyValueInfo analysis, which provides value range information

Loop



LoopSimplify Form



Red Hat

SimpleLoopUnswitch

```
while (...) {
  if (c) {
    foo();
  } else {
    bar();
  }
}
```

→

```
if (c) {
  while (...) {
    foo();
  }
} else {
  while (...) {
    bar();
  }
}
```