A whirlwind tour of the LLVM optimizer

Nikita Popov  @  EuroLLVM 2023
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](https://llvm.org/))
...ends

Frontend → Middle-end → Backend

Clang
Rust
Swift
Julia
...

X86
AArch64
ARM
RISCV
...

...
Default (non-LTO) pipeline

Module 1 → Optimize → Module 1'

Module 2 → Module 2'

Module 3 → Module 3'
Full LTO pipeline

Module 1 → Module 1'

Module 2 → Module 2'

Module 3 → Module 3'

Merge

Module M

Post-link optimize

Module M'
Thin LTO pipeline

Module 1 → Module 1' → Module 1'' → Module 1'''
Module 2 → Module 2' → Module 2'' → Module 2'''
Module 3 → Module 3' → Module 3'' → Module 3'''
Default pipeline

Module Simplification → Module Optimization → Backend
Default pipeline

More canonical

Module Simplification → Module Optimization → Backend

Less canonical
Default pipeline

More canonical

Module Simplification → Module Optimization → Backend

Less canonical

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

More canonical

Module Simplification

Inlining
Mem2Reg
LICM
...
Make further opts easier

Module Optimization

Vectorization
Runtime unrolling
...
Make further opts harder

Backend

Less canonical
Default pipeline

Module Simplification → Module Optimization → Backend

More canonical

Inlining
Mem2Reg
LICM
...
Make further opts easier

Vectorization
Runtime unrolling
...
Make further opts harder

Less canonical

Target-specific optimization
Lowering to machine code
ThinLTO pipeline

Module 1
Simplification

Module 1'
Simplification

Module 2
Simplification

Module 2'
Simplification

Module 1'
Optimization

Module 2'
Optimization

Cross
import

Pre-link

Post-link
ThinLTO pipeline

Module 1
Simplification

Module 1'
Simplification

Module 1'
Optimization

Module 2
Simplification

Module 2'
Simplification

Module 2'
Optimization

Cross import

Pre-link

Post-link

Second round of inlining
ThinLTO pipeline

Module 1
Simplification

Module 2
Simplification

Module 1'
Simplification

Module 2'
Simplification

Module 1'
Optimization

Module 2'
Optimization

Cross import

Second round of inlining

Pre-link

Post-link

Don't run decanonicalizing transforms pre-link
Module Simplification

CGSCC Pipeline

Early Cleanup → Inlining Function Simplification → Late Cleanup
CGSCC Pipeline
CGSCC Pipeline

![Graph Diagram]

- **f** → **g**
- **g** → **h**
- **g** → **i**

**simplify**

Red Hat
CGSCC Pipeline

g,h

f -> g,h
try inline

i
simplify
CGSCC Pipeline

- f -> g,h
  - try inline simplify
  - i -> simplify
CGSCC Pipeline

try inline simplify

\[ f \rightarrow g, h \]

try inline simplify

\[ g, h \rightarrow i \]

simplify
CGSCC Pipeline

Inlining sees already simplified functions!
Call-Graph Strongly Connected Components

SCC 3  SCC 2  SCC 1
\[ \text{SCC 1} \]

No well-defined order within SCC
Running pipelines

- `opt -passes='default<03>'` == `opt -O3`
- `opt -passes='thinlto-pre-link<03>'`
- `opt -passes='thinlto<03>'`
- `opt -passes='lto-pre-link<03>'`
- `opt -passes='lto<03>'`
opt -passes='default<O3>' -print-pipeline-passes

Defined in PassBuilderPipelines.cpp
godbolt.org – LLVM Opt Pipeline
godbolt.org – LLVM Opt Pipeline
Or run `opt -print-after-all` locally.
SSA Construction
Mem2Reg

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

```mlang
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();
}
SimplifyCFG: Speculate

```c
if (cond) {
    x = foo();
tmp = foo();
    x = cond ? tmp : 0;
} else {
    x = 0;
}
```
SimplifyCFG: Switch to lookup table

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

```c
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 → InstSimplify → ConstantFolding
InstCombine: Analysis helpers

- ConstantFolding
  - Folds instructions with constant operands to constants
  - $1 + 2 \Rightarrow 3$
InstCombine: Analysis helpers

- **ConstantFolding**
  - Folds instructions with constant operands to constants
  - $1 + 2 \Rightarrow 3$

- **InstSimplify**
  - Folds instructions to existing values or constants
  - $x + 0 \Rightarrow x$
  - $x - x \Rightarrow 0$
InstCombine: Analysis helpers

- **ConstantFolding**
  - Folds instructions with constant operands to constants
  - $1 + 2 \Rightarrow 3$

- **InstSimplify**
  - Folds instructions to existing values or constants
  - $x + 0 \Rightarrow x$
  - $x - x \Rightarrow 0$

- **InstCombine**
  - Tries constant folding and instruction simplification first
  - Performs folds that create or modify instructions
  - $x * 4 \Rightarrow x \ll 2$
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`
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);
```

```diff
add1 = x + y;
// ...
use(add1);
```
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);
GVN: Global Value Numbering

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

- Optimize memcpy and memset using MemorySSA
MemCpyOpt: Memcpy forwarding

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

```c
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 = v2;
DSE: Dead before return

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

```c
%p = alloca i32 
; ... 
; %p not read here 
ret void
```
Loop Optimization
Loop pass manager

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

Preheader

Loop

\[
\begin{align*}
&x = \text{foo}() \\
&\text{use}(x) \\
&y = \text{bar}() \\
\end{align*}
\]

Exit

\[
\begin{align*}
&\text{use}(y)
\end{align*}
\]
LICM: Hoist

Preheader

\[ x = \text{foo}(); \]

Loop

\[ \text{use}(x); \]
\[ y = \text{bar}(); \]

Exit

\[ \text{use}(y); \]
LICM: Sink

Preheader

\[ x = \text{foo}(); \]

Loop

\[ \text{use}(x); \]
\[ y = \text{bar}(); \]

Exit

\[ \text{use}(y); \]
LICM: Sink

Preheader

\[ x = \text{foo}(); \]

Loop

\[ \text{use}(x); \]

Exit

\[ y = \text{bar}(); \]
\[ \text{use}(y); \]
LICM: Promote

Preheader

Loop

\[
\begin{align*}
v &= *p; \\
v_n &= v + 1; \\
*p &= v_n;
\end{align*}
\]

Exit
LICM: Promote

Preheader

\[ v_0 = *p; \]

Loop

\[ v = \phi(v_0, vn); \]
\[ vn = v + 1; \]

Exit

\[ *p = vn; \]
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

```c
unsigned test(unsigned n) {
    unsigned sum = 0;
    for (unsigned i = 0; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```
IndVarSimplify: Loop exit value replacement

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

```c
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++) {}  // Later removed by LoopDeletion
    return (n * (n - 1))/2 + n;
}
LoopUnroll: Full unrolling
LoopUnroll: Loop peeling

Iteration #1-N

Iteration #1

Iteration #2-N
LoopUnroll: Partial unrolling

- Iteration #1-400
- Iteration #(4i+1)
  - Iteration #(4i+2)
  - Iteration #(4i+3)
  - Iteration #(4i+4)
LoopUnroll: Runtime unrolling

- Iteration #1-N
  - Iteration #(4i+1)
  - Iteration #(4i+2)
  - Iteration #(4i+3)
  - Iteration #(4i+4)
  - Tail iterations
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
```python
if (x > 10) {
    greater10();
}
always();
if (x > 0) {
    greater0();
}
else {
    always();
}
```
JumpThreading

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

Diagram:

- Preheader
- Header
- Latch
- Loop
- Backedge
- Exit 1
- Exit 2
SimpleLoopUnswitch

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

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