Code Generation for Data Processing

Lecture 6: Optimizations in LLVM

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2025/26

Optimizations in LLVM

- Analysis: collect information about IR
  - Examples: loop info, alias analysis, branch probabilities
  - Results are cached, re-run when outdated
- Pass: perform transformation on module/CGSCC/function/loop
  - Examples: instruction combine, simplify CFG, global value numbering, scalar replacement of aggregates (SROA; promotes `alloca` to SSA)
  - Input and output IR must be valid
  - Can use results of analyses; invalidates (some) analyses on changes
- Pass Manager: execute series of passes of same granularity
  - Otherwise, use adaptor: `createFunctionToLoopPassAdaptor`

LLVM Optimization Pipeline (Simplified)[31]

Module Pipeline (`buildPerModuleDefaultPipeline`): two parts

Module Simplification Pipeline (`buildModuleSimplificationPipeline`)

- Early function simplification (SimplifyCFG, SROA, EarlyCSE)
- CGSCC (post-order) (repeated after devirtualization):
  - Inliner
  - Function Simplification Pipeline (`buildFunctionSimplificationPipeline`)
    - SROA, SimplifyCFG, InstCombine, Reassociate, GVN, DCE, LICM
    - Most passes are here

Module Optimization Pipeline (`buildModuleOptimizationPipeline`)

- Primarily hard-to-reverse loop optimizations that are not simplifications
- E.g. vectorization, loop unrolling, loop distribution

[31] See `llvm/lib/Passes/PassBuilderPipelines.cpp` for full details.

SSA Construction: Mem2Reg and SROA

- Front-ends typically emit `allocas` for variables
  - Easier; also simplifies debugging and debug information
- Mem2reg: promote `alloca` to SSA values/phis
  - Condition: only `load`/`store`, no address taken
  - Essentially just SSA construction
- SROA: scalar replacement of aggregate
  - Separate structure fields into separate variables
  - Also promote them to SSA, subsumes mem2reg

Simplification

- Many operations can be expressed in various ways
  - E.g.: `sub i32 %x, 1`, `add i32 %x, -1`, `add i32 -1, %x`
- Problem: always need to consider all equivalent cases

$\Rightarrow$ Canonicalize IR: one single preferred form
  - Single instructions: e.g. operand order
  - Instruction sequences: e.g. series of additions, useless PHIs
  - CFG: e.g. branches, loop layout
- E.g.: constants go to the right, constant `sub` is `add`,
  loops with fixed trip count turned into `for (i = 0; i != 25; ++i)`, ...

InstCombine

- Main instruction canonicalization and simplification pass
- Includes many arithmetic-logical patterns, folding of PHIs, ...
- Implementation:
  - Add instructions to work list
    - Trivially dead/constant instructions eliminated immediately
    - Blocks traversed in reverse post-order
  - Patterns applied to instructions; on match, replace all uses
  - Changed instruction re-enqueues all users to work list
- Historically: fixed-point iteration, now just one iter. per run
- Covers many patterns, many patterns missing, >50kLOC C++

IR Pattern Matching

- IR patterns are typically DAG matches on def–use graph
- Typically implemented via helper functions[32]

```
// Excerpt from InstCombinerImpl::visitAdd

// (add (xor A, B) (and A, B)) --> (or A, B)
// (add (and A, B) (xor A, B)) --> (or A, B)
if (match(&I, m_c_BinOp(m_Xor(m_Value(A), m_Value(B)),
                        m_c_And(m_Deferred(A), m_Deferred(B)))))
  return BinaryOperator::CreateOr(A, B);
```

[32] llvm/IR/PatternMatch.h

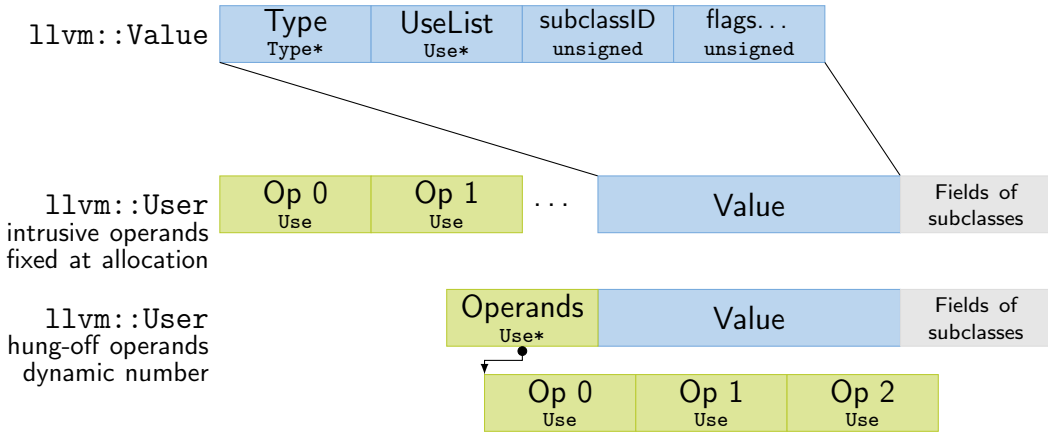
LLVM: Use Tracking

- Values track their users

```
llvm::Value* v = /* ... */;
for (llvm::User* u : v->users())
  if (auto i = llvm::dyn_cast<llvm::Instruction>(u))
    // ...
```

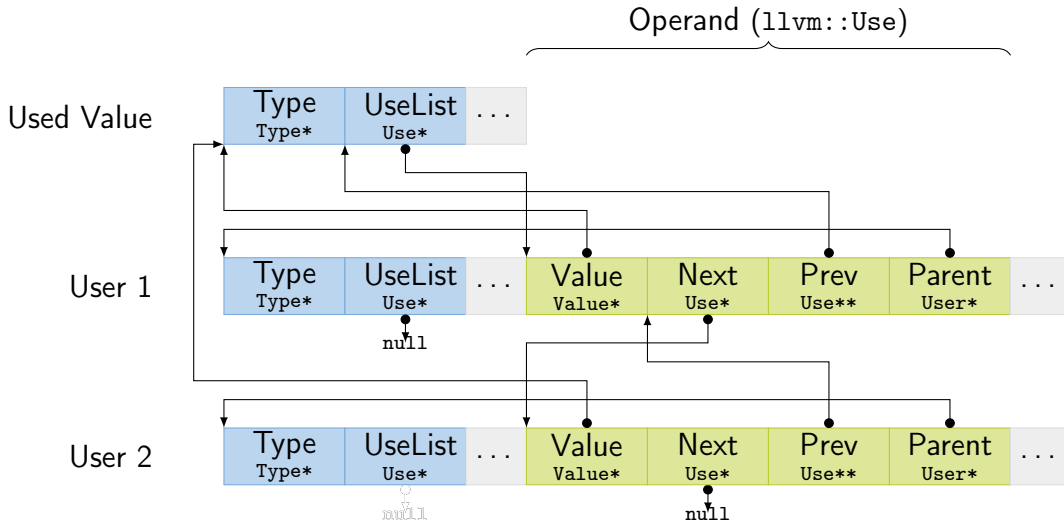
- Allows for easy replacement:
  - `inst->replaceAllUsesWith(replVal);`

LLVM IR Implementation: Value/User



`PHINode` additionally stores *n* `BasicBlock*` after the operands, but aren't users of blocks.

LLVM IR Implementation: Use



LLVM IR Implementation: Instructions/Blocks

- `Instruction` and `BasicBlock` have pointers to parent and next/prev
  - Linked list updated on changes and used for iteration
  - Instructions have cached *order* (integer) for fast "comes before"
- `BasicBlock` successors: blocks used by terminator
- `BasicBlock` predecessors:
  - Iterate over users of block – these are terminators (and `blockaddress`)
  - Ignore non-terminators, parent of using terminator is predecessor
  - Same predecessor might be duplicated ($\rightsquigarrow$ `getUniquePredecessor()`)
- Finding first non-ϕ requires iterating over ϕ-nodes

Valid Transforms?

Which of these transformations src→tgt are valid?

```
define float @src1(float %x) {
  %a = fadd float %x, 0.0
  ret float %a
}
```

```
define float @tgt1(float %x) {
  ret float %x
}
```

```
define i1 @src2(i8 %x) {
  %xm1 = add i8 %x, -1
  %y = xor i8 %x, %xm1
  %r = icmp ule i8 %x, %y
  ret i1 %r
}
```

```
define i1 @tgt2(i8 %x) {
  %ctpop = call i8 @llvm.ctpop(i8 %x)
  %r = icmp ult i8 %ctpop, 2
  ret i1 %r
}
```

```
define i8 @src3(i8 %x) {
  %div = sdiv i8 %x, 4
  ret i8 %div
}
```

```
define i8 @tgt3(i8 %x) {
  %div = ashr i8 %x, 2
  ret i8 %div
}
```

Alive2

- Seemingly correct transformations might be wrong in edge cases
- Unexpected transformations can be correct

$\Rightarrow$ Prove correctness of transformations

- Alive2[33]: translation validation tool for LLVM-IR
- Convert LLVM-IR into SMT formulas
- Proves that target is a refinement of source function

- InstCombine transformations require Alive2 proofs nowadays[34]

[33] NP Lopes et al. "Alive2: bounded translation validation for LLVM". In: *PLDI*. 2021, pp. 65–79.

[34] Random Example: `https://alive2.llvm.org/ce/z/xe_vb2`

Value Tracking

- Many transformations benefit from value ranges and known bits
  - E.g., fold `trunc`+shift into smaller shift if shift amount is known small
- `computeKnownBits`: recursively collect information from instrs
  - Arithmetic-logical instrs, casts, several intrinsics, etc.

- Bitwise instructions easier to reason about $\rightsquigarrow$ preferred canonicalization
  - E.g. `(1 << X) - 1` $\rightarrow$ `(-1 << X) ^ -1`
  - E.g. `sext(X)` $\rightarrow$ `zext(X)` if `X` is known non-negative

- Also use knowledge from front-end (but how encoded?)

Attribute, Flags, Assume

- Paramter/return attributes: constrain value ranges
  - `range(from, to)`, `nonnull`, `noundef`, ...
  - Values outside of the range are `poison`
- Instruction flags: encode additional knowledge about operation
  - `add/sub/mul/trunc` `nsw/nuw`: no signed/unsigned wrap
  - `or` `disjoint`: combination of disjoint bits
  - `udiv/sdiv/lshr/ashr` `exact`: divisor multiple of dividend
  - `getelementptr` `nuw/nusw/inbounds`: no wrap/stays inside allocation
  - `zext/uitofp` `nneg`: operand is non-negative
  - `icmp` `samesign`: operands have the same sign
  - Violation is `poison`

Assume

- Intrinsic `llvm.assume(i1)`
- Conditional immediate undefined behavior if argument is `false`
- Requires extra instruction for condition $\rightsquigarrow$ might be problematic
  - Instructions use other values, etc. $\rightsquigarrow$ prevent optimizations
- Also supports operand bundles for `nonnull`, `align`, `separate_storage`

Valid Transforms?

Which of these transformations src→tgt are valid?

```
define i8 @src1(i8 %x) {
  %div = udiv exact i8 1, %x
  ret i8 %div
}
```

```
define i8 @tgt1(i8 %x) {
  ret i8 1
}
```

```
define i1 @src2(i8 %x, i8 %y) {
  %cmp = icmp sge i8 %x, %y
  %cmpeq = icmp samesign eq i8 %x, 127
  %r = select i1 %cmp, i1 %cmpeq, i1 false
  ret i1 %r
}
```

```
define i1 @tgt2(i8 %x, i8 %y) {
  %cmpeq = icmp samesign eq i8 %x, 127
  ret i1 %cmpeq
}
```

```
define i8 @src3(i8 %x, i8 %y, i8 %z) {
  %xy = add nsw i8 %x, %y
  %xyz = add nsw i8 %xy, %z
  ret i8 %xyz
}
```

```
define i8 @tgt3(i8 %x, i8 %y, i8 %z) {
  %xz = add nsw i8 %x, %z
  %xyz = add nsw i8 %xz, %y
  ret i8 %xyz
}
```

Fast-Math Flags

- Floating-point arithmetic typically follows IEEE 754
  - Except for NaN, which is more relaxed
- Implication: almost impossible to optimize
  - non-associative, NaNs, infinity, negative zero, no reciprocals, no contraction
  - Transformations can result in vastly different results!
- Additionally: floating-point exceptions
  - Typically ignored, but some users might be interested

- Value/operation flags to permit transformations locally
- `strictfp` function attribute allows more control through constrained floating-point intrinsics

Branch Weights

- Annotation of branch with expected probabilities
  - Sources: programmer annotations, execution profiles
- Added as metadata to branch instructions
- Can compute block execution frequencies

```
define i32 @f(i32 %x) {
  ; ...
  %cmp = icmp eq i32 %x, 0
  br i1 %cmp, label %then, label %else, !prof !5
  ; ...
}

!5 = !{!"branch_weights", !"expected", i32 1, i32 2000}
```

Function Argument Optimizations

- Inferring argument attributes
- Eliminate dead arguments
  - Optimistic liveness analysis of arguments (assume dead)
  - Create new function with fewer arguments
- Promote pointer arguments to value arguments
  - Esp. relevant for C++ pass-by-reference
  - Only possible when both caller/callee CPU features match

Other Inter-Procedural Optimizations

- Inlining / Outlining
- Function Specialization
- Optimization of global variables
  - Marking as constant, delete dead variables, attributes, etc.
- Inter-procedural SCCP
- Devirtualization / call-target speculation
- Hot-Cold Splitting
- ...

Optimization Remarks

- Understanding optimization decisions can be difficult
  - Complex code base, abstraction, heuristics, ...
- Optimization remarks: passes can report information
  - Inlining decisions: provide cost values/thresholds for decisions
  - Others: GVN, LICM, FastISel
- Output also provided as YAML or a binary format for analysis
- Lots of verbose output, hard to identify important points

- Clang: `-Rpass=<pass>` `-Rpass-analysis=<pass>` `-Rpass-missed=<pass>`

Other Aspects not Covered Here

- Analyses: alias analysis, MemorySSA, Scalar Evolution (SCEV), target-specific analyses/heuristics, ...
- Optimizations: various loop transformations (distribute, fuse, flatten, polyhedral optimizations), value propagation, vectorization, reassociation, ...
- Loop-Closed SSA canonical form
  - Values defined inside loop only used in loop or in ϕ-node in exit
  - Simplifies tracking uses of values defined in loops

Using LLVM (New) Pass Manager

```
void optimize(llvm::Function* fn) {
  llvm::PassBuilder pb;
  llvm::LoopAnalysisManager lam{};
  llvm::FunctionAnalysisManager fam{};
  llvm::CGSCCAnalysisManager cgam{};
  llvm::ModuleAnalysisManager mam{};
  pb.registerModuleAnalyses(mam);
  pb.registerCGSCCAnalyses(cgam);
  pb.registerFunctionAnalyses(fam);
  pb.registerLoopAnalyses(lam);
  pb.crossRegisterProxies(lam, fam, cgam, mam);

  llvm::FunctionPassManager fpm{};
  fpm.addPass(llvm::DCEPass());
  fpm.addPass(llvm::createFunctionToLoopPassAdaptor(llvm::LoopRotatePass()));
  fpm.run(*fn, fam);
}
```

Writing a Pass for LLVM's New PM – Part 1

```
#include "llvm/IR/PassManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"

class TestPass : public llvm::PassInfoMixin<TestPass> {
public:
  llvm::PreservedAnalyses run(llvm::Function &F,
                              llvm::FunctionAnalysisManager &AM) {
    // Do some magic
    llvm::DominatorTree *DT = &AM.getResult<llvm::DominatorTreeAnalysis>(F);
    // ...
    llvm::errs() << F.getName() << "\n";
    return llvm::PreservedAnalyses::all();
  }
};
// ...
```

Writing a Pass for LLVM's New PM – Part 2

```
extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
llvmGetPassPluginInfo() {
  return { LLVM_PLUGIN_API_VERSION, "TestPass", "v1",
    [] (llvm::PassBuilder &PB) {
      PB.registerPipelineParsingCallback(
        [] (llvm::StringRef Name, llvm::FunctionPassManager &FPM,
            llvm::ArrayRef<llvm::PassBuilder::PipelineElement>) {
          if (Name == "testpass") {
            FPM.addPass(TestPass());
            return true;
          }
          return false;
        });
    } };
}
```

```
c++ -shared -o testpass.so testpass.cc -lLLVM -fPIC
opt -S -load-pass-plugin=$PWD/testpass.so -passes=testpass input.ll
```

Optimizations in LLVM – Summary

- LLVM provides a wide range of analyses and optimizations
- Optimizations organized in flexibly composeable passes
- Simplification centered around canonicalization
- Much knowledge inferred from assumptions (and `poison`/UB)
- SSA simplifies many transforms through explicit def–use chains
- Only few non-simplifying transformation passes

Optimizations in LLVM – Questions

- Why are some passes executed multiple times during optimization?
- How does the simplification pipeline relate to inlining?
- What is the key benefit of canonicalizing the IR?
- How does LLVM's `replaceAllUsesWith` work?
- What is the benefit of instr. flags like `nsw`?
- Why are floating-point optimizations problematic?
- How is C++ `[[likely]]` represented in LLVM-IR?